NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DATA MINING OF EXTREMELY LARGE AD-HOC DATA SETS TO PRODUCE REVERSE WEB-LINK GRAPHS

by

Tao-hsiang Chang

March 2017

Thesis Co-Advisors: Frank Kragh
Geoffrey Xie

Approved for public release. Distribution is **unlimited.**

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2017	Master's Thesis 12-28-2015 to 03-31-2017

4. TITLE AND SUBTITLE

DATA MINING OF EXTREMELY LARGE AD-HOC DATA SETS TO PRODUCE REVERSE WEB-LINK GRAPHS

5. FUNDING NUMBERS

6. AUTHOR(S)

Tao-hsiang Chang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Data mining can be a valuable tool, particularly in the acquisition of military intelligence. As the second study within a larger Naval Postgraduate School research project using Amazon Web Services (AWS), this thesis focuses on data mining on a very large data set (32 TB) with the open web crawler data set Common Crawl. Similar to previous studies, this research employs MapReduce (MR) for sorting and categorizing output value pairs. Our research, however, is the first to implement the basic Reverse Web-Link Graph (RWLG) algorithm as a search capability for web sites, with validation that it works correctly. A second goal is to extend the RWLG algorithm using a full Common Crawl archive as input for processing as a single MR job. To mitigate the out-of-memory error, we relate some environment variables with the Yet Another Resource Negotiator (YARN) architecture and provide some sample error tracking methods. As a further contribution, this study considers limitations associated with using AWS, which inform our recommendations for future work.

14. SUBJECT TERMS

Amazon Web Services, cluster computing, data mining, Hadoop MapReduce, the Common Crawl

15. NUMBER OF PAGES 119

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DATA MINING OF EXTREMELY LARGE AD-HOC DATA SETS TO PRODUCE REVERSE WEB-LINK GRAPHS

Tao-hsiang Chang
Lieutenant, Taiwan Navy
B.S., The Citadel, 2009

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2017

Approved by: Frank Kragh
Thesis Co-Advisor

Geoffrey Xie
Thesis Co-Advisor

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Data mining can be a valuable tool, particularly in the acquisition of military intelligence. As the second study within a larger Naval Postgraduate School research project using Amazon Web Services (AWS), this thesis focuses on data mining on a very large data set (32 TB) with the open web crawler data set Common Crawl. Similar to previous studies, this research employs MapReduce (MR) for sorting and categorizing output value pairs. Our research, however, is the first to implement the basic Reverse Web-Link Graph (RWLG) algorithm as a search capability for websites, with validation that it works correctly. A second goal is to extend the RWLG algorithm using a full Common Crawl archive as input for processing as a single MR job. To mitigate the out-of-memory error, we relate some environment variables with the Yet Another Resource Negotiator (YARN) architecture and provide some sample error tracking methods. As a further contribution, this study considers limitations associated with using AWS, which inform our recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AWS Amazon Web Services

BNF Backus–Naur Form

CLI Command Line Interface

EC2 Elastic Compute Cloud

EMR Elastic MapReduce

GUI Graphic User Interface

HDFS Hadoop File System

HMR Hadoop MapReduce

HTML HyperText Markup Language

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IRI Internationalized Resource Identifier

JAR Java Archive

JSON JavaScript Object Notation

MR MapReduce

NPS Naval Postgraduate School

PaaS Platform as a Service

RDD Resilient Distributed Dataset

RFC Request for Comments

RWLG Reverse Web-Link Graph

S3 Simple Storage Service

SaaS Software as a Service

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

WARC Web ARChive

WWW World Wide Web

YARN Yet Another Resource Negotiator

Acknowledgments

First, I want to thank my institution and the U.S. Navy, which offered me the chance to attend Naval Postgraduate School and accomplish many of my personal goals. Second, I am grateful for modern-day technology, which allowed me to feel close to my family and receive their supportive words from thousands of miles away. In particular, I appreciate the dedication of my significant other. Without her love and commitment, I could never have finished either my master's degrees or this thesis. Most importantly, I would like to thank my advisors, Professor Frank Kragh and Professor Geoffrey Xie. The process would have been much more difficult without their direction and guidance.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Data mining is an interesting field of study as well as a popular tool for many organizations, and the military is no exception. The fact that the military values intelligence acquisition as an extremely important asset heightens the significance of data mining even more. Thanks to the growth of the Internet and the use of smart phones, military activities, for example, can be easily observed, recorded, or photographed by military personnel, their family and friends, and others. This information is easily posted on any social media site (e.g., Facebook, Twitter) and can be handily gathered and analyzed by anyone. As a result, the movement of troops is almost impossible to hide nowadays. Indeed, the amount of information released through social media every day is so enormous that no one individual can extract specific useful information on his own. For example, the average 350 million photos uploaded to Facebook each day [1] are nearly impossible to analyze and extract useful information from in a reasonable time. The photos themselves are already a tremendous source of information, not to mention videos and text posts. This is where data mining comes into play.

This is the second study within a larger Naval Postgraduate School (NPS) research project involving data mining with the open web crawler data set Common Crawl. The first study within the NPS data mining project was published in June 2016 by A. Coudray [2]. The topic for the present thesis was prompted by the Reverse Web-Link Graph (RWLG) algorithm in a paper written by Google scientists [3]. As a part of the larger NPS research project, the study described in this thesis shares the same services and tools, including Amazon Web Services (AWS), which offers various on-line computing services and management tools. Like the previous work [2], we use the Common Crawl data set as the input source for processing. We also use AWS Simple Storage Service (S3) for result storage and AWS Elastic MapReduce (EMR) and Elastic Compute Cloud (EC2) for data mining specific software and hardware. Essential background information to understand the rest of the chapters is provided within this work. Further details on Big Data, the Hadoop File System (HDFS), Hadoop MapReduce (HMR), AWS, and the Common Crawl can be found in [2].

1.1 Research Problem

The focus of this thesis is on processing an extremely large data set using the RWLG algorithm. In this study, RWLG is used to process the web pages as input, yielding ordered Uniform Resource Locator (URL) value pairs $< targetURL, sourceURL >$ which constitute a map showing how web pages link to one another. The usage of RWLG can be exemplified by the scenario that we only have someone's e-mail address, and we want to find out additional information about that person. We can use the hypothetical e-mail address as a key to find the corresponding links from the RWLG output-value pairs, and we can further look for those corresponding links to determine that person's job, company, community, or profession, yielding a description of the person of interest. We can even find further information about this person using the corresponding links as keys to find even more links.

As previously described, this research uses the RWLG algorithm, which extracts value pairs for one-hop links. We not only implement the RWLG algorithm but also extend the algorithm to extract link pairs for links of two or more hops under the MapReduce (MR) environment. In addition, this work considers the impact of invalid target URLs along with dynamic URLs, mitigates some problems encountered in previous works, and results in software capable of producing RWLGs on the scale of the Internet.

In this work, the emphasis is on dealing with an extremely large data set. As mentioned in the military activity observation example, various kinds of information can be extracted from the World Wide Web (WWW). The Common Crawl compiled its latest archive of all the accessible text portion of the Web pages into 57 TB of compressed files as of the first quarter of 2017. Certainly, data mining is used almost everywhere nowadays, but only a few applications are required to process this extreme amount of data. Coudray's thesis was a good starting point for this research project, but he encountered many issues while processing data that did not even approach one percent of the archive. This study tries different approaches to deal with a larger amount of data.

1.2 Organization

The literature review and background information necessary to understand this research are presented in Chapter 2. The tools used in the research, a guide to the environment setup, the

algorithms developed for this work, and proofs showing the validity of these algorithms are explained in Chapter 3. The results of this research are exhibited in Chapter 4. Finally, the work is summarized and possible avenues for future work are recommended in Chapter 5.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

To work with data mining and tap its efficiency, we need specific hardware and software. For hardware, a cluster of computers is preferred, and we can either set up our own cluster, or nowadays, rent one from a cloud services company. For software, we need a tool appropriate for cluster computing and parallel processing that is suitable for data mining. HMR, a module MR under the Apache Hadoop framework, is one of the typical tools for cluster computing and distributed processing. For this research, we used both hardware and software services provided by AWS, so we did not have to set up or maintain hardware or install software. The only effort required was to learn how to use their services, which is relatively straightforward. Besides hardware and software, we also needed a proper data set for data mining processing. The Common Crawl is a suitable data set on which to do different kinds of processing.

Since this study is part of a larger project, we share almost the same background with [2]. As the first work in the NPS research project, in the Coudray thesis Big Data and HMR infrastructure were described in detail from a very basic foundation [2]. This study, the second work in the research project, provides only background sufficient to understand this thesis. Although the HMR is described in [2], one of the key elements in MR version 2, Yet Another Resource Negotiator (YARN) [4], was not and is explained here. As an internal resource locator in HMR, YARN was very important in this study and is described with some MR characteristics. Spark, which was not used in this study, is usually compared with MR and, therefore, is also explained here. AWS, as the platform to execute our code, is discussed, focusing on its changes from the time when the Coudray thesis was published. The Common Crawl, as the source data set, is also discussed and compared with its usage in Coudray's work [2]. URLs, which play an important role in the RWLG algorithm in this thesis, are yet another topic discussed in this chapter. Finally, we end this chapter with a comparison of this study to previous research.

2.1 MR

As stated in Chapter 1, the research idea stems from Google's MR paper [3]. MR is a module in the Apache Hadoop framework. To work with MR requires thinking about processing data in a special way: make everything into value pairs that can be easily processed in parallel. MR gets its name from the symbolic two sequential stages: Map and Reduce. Mappers are the distinct processes that run the same algorithm in the Map stage, as are the reducers in the Reduce stage. The source data set is separated into different blocks for different mappers to process and produces output as $< key, value >$ value pairs. These value pairs are transferred to reducers by key through a step called shuffle and sort. Each reducer then handles these pairs of a single key and produces the final value pairs for the key. MR is a simple tool that is easy to learn, write, and deploy, but which has also proved to be inefficient for many kinds of applications; hence, the next generation, Spark, was invented [5].

2.1.1 YARN in MR Version 2

HMR is capable of doing parallel computing on clusters, and YARN is a core element in MR that manages the resources on all clusters. From the description of the MapReduce official website [6], we notice the key parts of YARN: ResourceManager, ApplicationMaster, Node Manager, and Container. YARN architecture is illustrated in Figure 2.1.

2.1.2 YARN Architecture

We can see that one node runs the ResourceManager, and other nodes have a NodeManager per node. ApplicationMasters and Containers reside in the cluster nodes, and the ApplicationMaster is in charge of its Containers. The mappers and reducers we are familiar with run in these Containers. Understanding the YARN architecture and knowing the relation between its parts is critical when setting the MR environment variables. The amount of memory in a Container limits the number of mappers and reducers that can be accommodated. The amount of memory in a node limits the number and size of the Containers that can be accommodated. Without this fundamental knowledge, one gets lost when setting up the memory parameters.

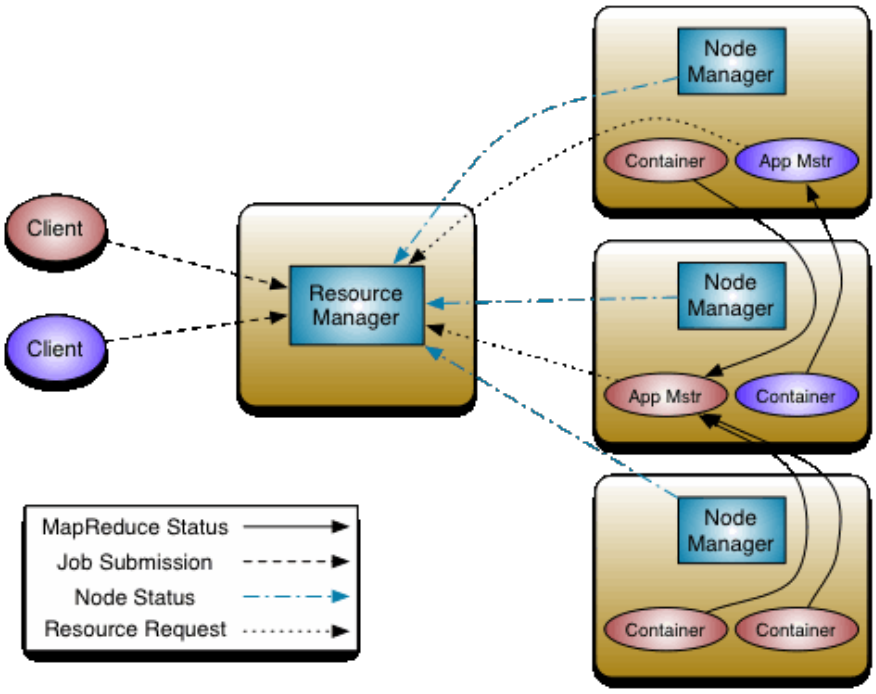


Figure 2.1. Description of YARN Architecture. Source: [6].

2.1.3 Memory Parameter Setting

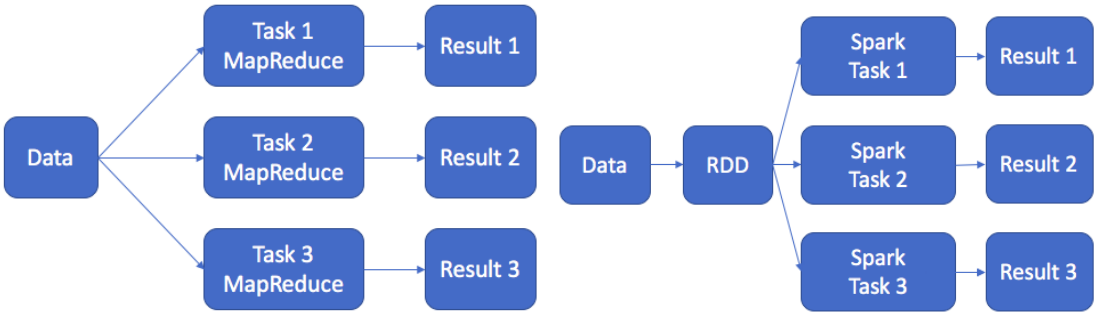
Other than introducing the MR infrastructure and mechanisms, [7] provides important YARN properties and some assumptions regarding memory settings. Since we are using EMR, we do not need to set up the YARN architecture, but we do need to set up the memory portion, a discuss [7] does not provide. Fortunately, [8] offers a set of equations for calculating a "rule of thumb" memory setting depending on the specifications of the cluster, which builds on top of an understanding of the YARN architecture. These introductory resources gave us the initial guidelines to adjust the memory setting and served as a good starting point in our research.

2.2 Spark

As mentioned earlier, Spark, which is similar to MR, is another tool for cluster computing. Spark is known to be more efficient than MR in many types of applications by introducing Resilient Distributed Dataset (RDD) [9]; thus, it was an option while starting this research.

As there is no cache mechanism in MR, when an iterative job such as linear regression is

required, MR must start again from the very beginning in every cycle, causing poor MR performance. MR's performance suffers as it must launch the whole process again in every task, as shown in Figure 2.2a.



(a) Iterative Job Operation for MR (b) Iterative Job Operation for Spark

Figure 2.2. Iterative Job Operation Figures for MR and Spark

We can think of RDD as a cache that saves the mapper's output. Once the job requires the previously computed value pairs, it checks the RDD first before starting the Map stage to run the whole data set again and, thus, saves a significant amount of execution time. Spark launches the process only once, and every task uses the same computed values from RDD, as shown in Figure 2.2b.

Within this study's analysis, however, the RWLG is not found by an iterative job; it does not require the computed data to be reused before emitting its output, at least not when the algorithm is created. It is more likely to have massive sorting, which is the only application for which MR is superior to Spark [10], [11]. MR was, therefore, chosen to be the tool for this research.

2.3 AWS

AWS provides various online services in three different types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). For the services of interest in this work, EC2 and S3 belong to IaaS, and EMR belongs to PaaS. There are actually two ways to run HMR in AWS: directly use the EMR platform provided or to use multiple instances from EC2 and build from the infrastructure. In both [2] and this thesis, EMR is used directly because it is not only cheaper but also enables us to reduce the complexity of building the infrastructure. Nevertheless, we do not know how Amazon has

configured the lower levels, which causes problems when conducting our research. Note that EMR runs on EC2 instances, while EC2 is transparent to EMR users. These two services, though, do share the same limitation, and the EMR user still needs to examine the EC2 documents when errors occur.

The AWS environment changed a lot in nine months, which was the period between the publication of [2] and this work. There are not many conceptual changes, but a lot of changes were made to the functions and mechanisms by updates and version changes in a popular programming environment. General changes about AWS, working environment, tools, and pricing information are described separately in the following section.

2.3.1 General AWS Changes

Since both works run MR tasks, it makes sense that both works deploy these tasks using the EMR service from AWS. The EMR release version has changed from 4.0 to 5.0, which includes several changes in functions and mechanisms. The most important difference in this study is that the Common Crawl data can no longer be accessed by EMR if we do not set the appropriate region. In other words, the EMR region has to be set to "US EAST (Northern Virginia)" now, which is the same region in which the Common Crawl files reside, for the MapReduce tasks to work correctly.

2.3.2 AWS GovCloud (US)

Because NPS is a government organization, we can, and are required, to use the government cloud for research unless none exists, which was the case for the Coudray thesis. This is no longer true (since late 2016); thus, the research was moved to the AWS GovCloud (US). Because of the security requirements of the GovCloud, it is physically separated from the normal AWS Cloud. In the current AWS system settings, the separation means the GovCloud is unable to access Common Crawl files, which reside in the normal AWS Cloud. To solve these issues, we established a special account that belongs to the government and has the same functionality as the normal AWS Cloud.

2.3.3 Command Line Interface

AWS's Graphic User Interface (GUI) was utilized in Coudray's work; however, this work involved many more experiments than Coudray's. The GUI would not work for this number of experiments required since it would take much more time to assign tasks. The Command Line Interface (CLI) is capable of accomplishing the same thing with a long, single command and can be scripted to start many different tasks quickly. This characteristic greatly improved the speed when executing experiments.

2.3.4 How to Choose Instances from AWS

Instances in AWS means machines that run our jobs. Performance is always a concern when running experiments, and AWS offers different kinds of instances with general guidelines in [12] for users to execute their tasks. We found publications using AWS, but none of them did a systematic comparison between these instances. There are studies comparing the performances between different numbers of instances used [13]–[15], but there are rarely comparisons between different types of instances used [15], [16]. As a result, it remained difficult to choose the specific type of instance that suits our research.

There are also works that examine resources and bottlenecks that constrain MR performance [17], [18]. In these studies, unexpectedly, we learned that the network capability is not the bottleneck in most of the MR cases. From these studies, we also learned that computing-optimized instances should be chosen for serialized/compressed input data processing from the Common Crawl, which is exactly our case. This assumption was tested and verified as discussed in later chapters.

2.3.5 Pricing Information

AWS gain their profit by providing services, and we, as a user, want to get the most out of the services while spending the least amount of money as well. There are considerations to think about while estimating the possible cost for a job and before executing the series of operations that we encountered.

There are three major sources to be considered while estimating the cost:

1. AWS EMR
   When we use EMR to execute our algorithm, we are charged for using it. The EMR provides a "box" for different types of instances, and the boxes are charged by hour. The charged amount depends on the types of instances the user is using, which is given by AWS in [19].
2. AWS EC2
   We note that EMR charges only for the box, not the instances. This is because the instance is being charged in the EC2 part, which the price is given by AWS in [20].
3. AWS S3 and Data Transfer
   The input and output for our study are both using S3. The input data set, the Common Crawl, is saved on the "US EAST (Northern Virginia)" region of S3, as stated in the previous section. The output of our program is saved on the S3 buckets that we create. The data saved on the S3 bucket is charged daily based on its size, so we want to discard the resulting output when it is large (e.g., the result from multiple segments as input) [21]. Furthermore, data transfer between a different region of S3 buckets is going to be charged as listed in the bottom part of [21] as well; thus, we want to set up our bucket and EMR in the same region as the Common Crawl to minimize the data transfer fee.

2.4 Common Crawl

The Common Crawl is a repository of periodic archives of the Internet. It uses S3 in AWS as storage space and has used a crawl engine to save copies of the Internet periodically since 2008. The recent format of the Internet archives is the Web ARChive (WARC) file format [22]. Every WARC file contains many WARC records, and each WARC record contains HyperText Markup Language (HTML) code of a single web page and some meta-data as a record header from the crawl engine.

For the current Common Crawl structure, an archive is a full copy of the Internet, which is separated into 100 segments, where each segment contains about 578 WARC files. Since each WARC file is about 1 GB, an archive is at least 57 TB. Such a large data set is an appropriate source data set for data mining and for this work. A view of the Common Crawl

directory using CLI is shown in Figure 2.3.

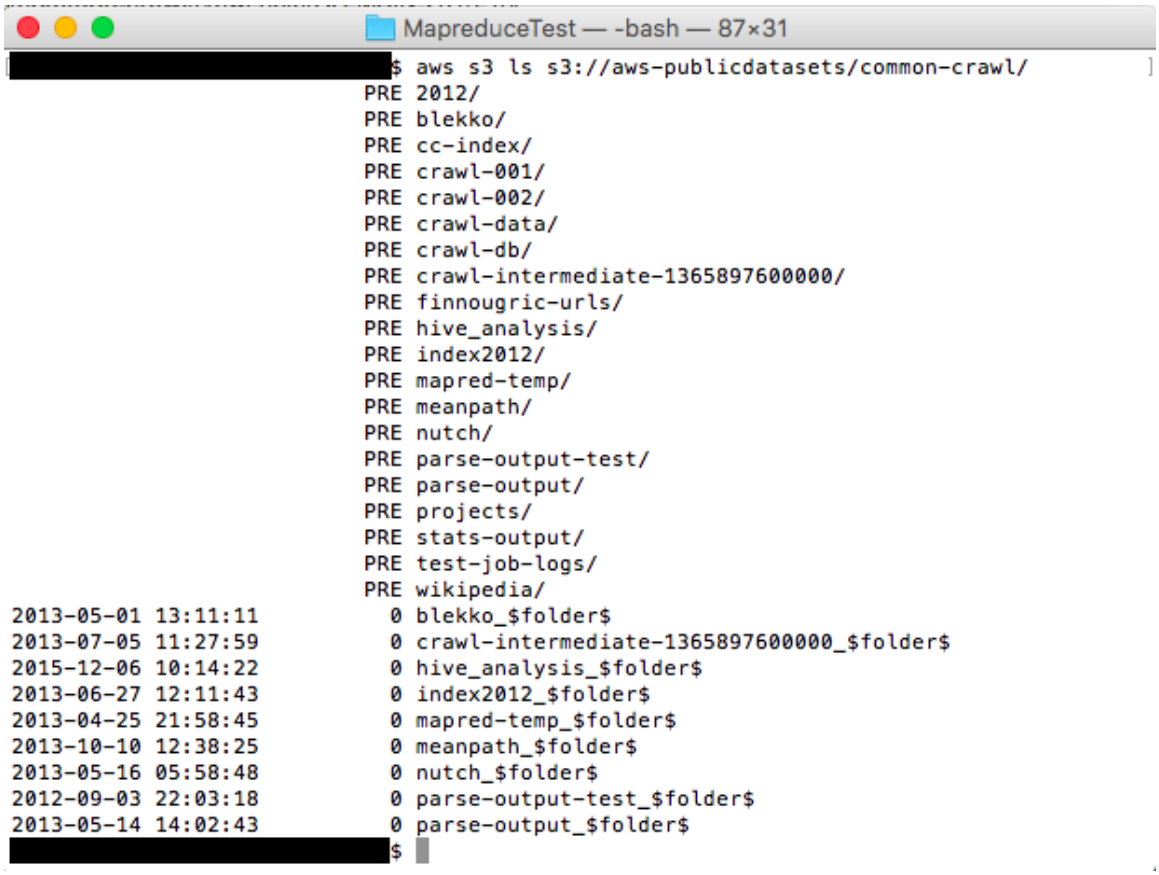


Figure 2.3. A View of the Common Crawl Directory Using CLI

2.5 URL

Understanding URLs is very important to this work since the output from the data mining in this research is nothing but URL value pairs. The regulations and definitions about these Internet-related concepts are regulated by documents called Requests For Comments (RFC). Before detailing the content of a URL, we should identify the difference between URL, Uniform Resource Name (URN), Uniform Resource Identifier (URI), and Internationalized Resource Identifier (IRI), documented in RFC1738, RFC3986, and RFC3987, respectively [23]–[25]. These three RFC documents describe the definitions and functionality of URL, URN, URI, and IRI, which are summarized in the following paragraph.

A URL is a specified resource locator. To access a resource indicated by a URL, one must

know its host's name, the directory in which the resource is stored, and the file name of the specific resource. The URL becomes useless if the corresponding resource is moved, which is a problem that resulted in the development of the URN. A URN looks like a URL, but the user only needs to know the resource name. By providing information such as a description of the resource along with the resource name, we get the most suitable copy of the resource from the server. URI is the superset of both URL and URN; that is, every URL is a URI, but a URI can be either a URL or a URN. The resource must be represented using the American Standard Code for Information Interchange (ASCII) character set, which causes a problem if the path or name of the resource contains foreign characters. IRI is intended to replace URI using a much wider character repository to "internationalize" it. In this work, we only focus on URLs that are already mature and widely used in the World Wide Web as well as in the Common Crawl.

2.5.1 URL Format

Since the URL is used to locate resources, its format, or scheme, varies depending on the type of resources. According to RFC1738, which is the document specifying URLs, we find there is a general format using the Backus–Naur form (BNF) representation:

<scheme>:<scheme-specific-part>.

The scheme is the type of applications used in the Internet, such as HTTP, FTP, etc. Since HTTP is the most common scheme encountered in this work, its BNF representation is described further as in the example:

http://<host>:<port>/<path>?<searchpart>

where <host> is the domain name in the Internet layer of the Internet protocol suite, and <port> is the port number in the Transport layer and is set to 80 by default if not omitted. The <path>, which is an optional HTTP selector that represents the file directory on the machine, and the <searchpart>, which is further described in Section 2.5.2 along with its preceding question mark, are optional as well. If all those optional parts are not used, the BNF representation can be simplified as follows:

http://<host>.

Note that characters like "/", ";", and "?" are reserved in the optional parts, and <searchpart> plays an important role in dynamic URLs.

2.5.2 Dynamic URL

A dynamic URL is a URL that contains the <searchpart> part. It is used to display specific parts of the content of that specific page, which usually contains a huge amount of data; however, the Common Crawl crawls these web pages as different pages, which increases the complexity of the resulting RWLG output. A dynamic URL is identified by the existence of a "?" in the BNF representation. This study provides the option to simplify the RWLG output by discarding the <searchpart> part, which removes the dynamic portion of the URL.

2.5.3 User-input-URL in HTML

There are two values in an output pair: source URL and target URL. Since we are using the Common Crawl data as an input source, the source URL is always extracted from the WARC record header in the WARC files, which are created by the crawl engine, so it is less likely to have problems. Nevertheless, the target URL is extracted from the HTML content created by other users; thus, having an invalid URL as the target URL is possible. Unfortunately, to validate each URL is very computationally expensive, so we did not try to validate URLs in this research. Actually, there are two types of invalid URLs: one is invalid from syntax. These errors occur very rarely and are categorized together by alphabetical order since they usually share the same set of special characters. The other type of invalid URL is known as link rot [26]. Link rot results in a "dead end" pair, which means this error does not propagate in our later tasks. It is not our focus to address either the broken links or URL validation, so no action was taken with regard to these invalid URLs.

2.6 Previous Research

Many students who are using AWS have also employed MR, but no research on RWLG has been published. By contrast, as the first step of the NPS research project, Coudray's study completes the Inverted Indices approach in the MapReduce paper, which takes the

web pages as input and emits $< word, list(sourceURL|position) >$ value pairs as output. Although the RWLG requires a different approach, this work still benefits from the tool-choosing process and shares some of the same problems addressed in Coudray's work, such as "OutOfMemory Error: Java heap space," and the AWS optimizing problem. Herein, we develop some solutions to increase our program's input size capability.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Experimental Design

The experimental design—from the choice of tools, environment setup, algorithms to experiment execution, especially focusing on the portions that are different from the previous work—is described in this chapter. The intent of this chapter is to give a clear and full understanding of how these experiments are designed and why they are designed this way, as well as how these experiments are conducted.

The organization of this chapter follows the research process: we first chose our tools and set up the environment for both local and cloud testing. Then, we designed our algorithms and implemented them into computer programs. Finally, we did local testing followed by cloud testing.

3.1 Tools

The previous chapters mentioned the key tools, such as AWS and the Common Crawl for cloud execution, but since launching a cloud execution is more expensive, we usually do a local test first. The local test can be run on almost any personal computer if the environment is properly configured.

As a single piece of a much larger NPS project, this study shares the same tool set as Coudray's [2], which is also part of that same project. Nevertheless, there are still some differences between our chosen tools and those of Coudray. Some of them are based on efficiency considerations, but most of them are just personal preferences.

Since we must write and pack our programs on our own machine, a handy Integrated Development Environment (IDE) like Eclipse is preferred. For local testing, Hadoop must be installed on the machine. For cloud execution, AWS must be set up properly. Both this and [2] use Eclipse as the IDE, but the way the Java code is packed into a compressed Java ARchive (JAR) is different. Both works require Hadoop on the local machine, but the install actions are different. Both studies use Common Crawl data for input data sets, but the archive chosen as the input source is distinct. Both efforts use AWS for cloud execution,

but the interfaces chosen to access AWS are different. These differences are all explained in the following sections.

3.1.1 Runnable JAR File Instead of Maven Project Packaging

In the HMR environment, our programs always had at least two separate Java files; thus, packaging them into one JAR file was essential. Maven is a powerful tool; it was used for packaging JAR files in Coudray's work [2]. Although it is a good tool, setting it up, including building the XML file, is complex just to package the JAR file. The current work used a much simpler way to achieve the same functionality without the need to install any plug-ins other than the main Eclipse IDE. The steps for using the built-in Eclipse function is described in later sections.

3.1.2 Native Hadoop Instead of VirtualBox Simulation

In Coudray's work, local testing was done by VirtualBox simulation [2]. He simplified his computer system this way while sacrificing execution efficiency. In contrast, this work gained more benefits by directly installing Hadoop on the host operating system.

First, deploying a virtual system requires system resources, and the execution is not able to run at full system capability. For the lightweight laptop that was used for this research, this distinction was important.

Second, VirtualBox uses more disk space than required. To handle the simulation output, we have to pre-allocate more resources for the virtual system, and the machine used cannot afford it.

Last, the file exchange is more complicated when using a virtual system. Without using a virtual system, we can pack the JAR file right at the execution directory and carry out testing with minimal delay. This is not possible in VirtualBox.

These considerations caused us to install Hadoop directly on the operating system. The installation of native Hadoop is discussed in later sections.

3.1.3 Newer Version of Common Crawl Files

The files in the Common Crawl are well organized; the numbers of WARC files in segments and the numbers of segments in archives are different. To be accurate, there are around 560 WARC files in each segment and 100 segments in an archive in the first 2017 archive. Coincidentally, Coudray's work [2] used a smaller-sized segment, but the current research used newer Common Crawl data sets. Although file size and total number of files should not be a problem in theory for parallel execution, we actually encountered hardware problems requiring mitigation. These are addressed in later sections.

3.1.4 Accessing AWS with CLI

GUIs are user friendly in most cases, including the AWS task configuration. The GUI is convenient and efficient when the number of jobs is small, but it can be time consuming to set up jobs using the GUI. The CLI is difficult to learn because it requires the user to learn many commands to carry out different functions, but it is much more efficient once the user learns these commands. In this research, there are many more jobs to run on different sets of data and instances than in the previous research; thus, using the CLI is arguably much more efficient. The environment setup for CLI and its usage on AWS S3/EMR is discussed in later sections.

3.2 Environment Setup

In this section, two different parts are contained: local machine setup and AWS configuration. Local machine setup includes the installation steps for Hadoop to a clean Mac OS, which can be used to write and run the MR jobs locally for basic testing and algorithm verification. AWS configuration describes the steps to set up privileges for CLI usage in the AWS as well as local setups such as account and region information, allowing direct execution of the AWS CLI commands from the Terminal application in the Mac OS.

3.2.1 Local Machine Setup

In this section, instructions to install Hadoop into a clean Mac OS, which requires specific software to be pre-installed, are contained. This instruction, integrating ideas from various

sources [7], [27], [28] with personal customization, aims to minimize the amount of human effort.

3.2.2 XCode

XCode is provided by Apple in the Mac App Store. It can be freely retrieved and should be installed as the very first item as it includes many functions and software that are transparent to the user and very useful in later steps.

3.2.3 Java

Java is provided by Oracle from its website (Java installation link can be found at https://www.java.com/) and can be retrieved at no cost. Java should be installed as the second item, since it is required to run Hadoop.

3.2.4 Eclipse

Eclipse is an IDE for many coding languages (Eclipse can be found at https://eclipse.org). Although Java code can be written using any text editor, an IDE helps greatly for formatting and syntax checking as well as for packaging. We simply download and install Eclipse through the GUI. There are seven libraries we must include: *hadoop-common* and *hadoop-mapreduce-client-core* are for Java programming involving Hadoop; *jwat-warc*, *jwat-common*, and *jwat-archive-common* are libraries that help the Java programmer to access WARC files; *warcutils* is a library that works with JWAT libraries to access Common Crawl WARC files; finally, the *jsoup* library is a Java HTML parser. The libraries *hadoop-common*, and *hadoop-mapreduce-client-core* are found at https://mvnrepository.com; the libraries *jwat-warc*, *jwat-common* and *jwat-archive-common* are found at https://sbforge.org/display/JWAT/JWAT; the library *warcutils* is found at https://github.com/norvigaward/warcutils; the *jsoup* library is found at https://jsoup.org/download.

3.2.5 Homebrew

Homebrew is third-party package manager software for the Mac OS (Homebrew installation guild can be found at http://brew.sh). The advantage of using Homebrew to install software

is that it identifies and installs all the dependencies, which simplifies the task when the user needs to install software under the Mac OS. Homebrew can be installed through a single line Terminal command after XCode is installed:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
    Homebrew/install/master/install)"
```

3.2.6 Hadoop

After the above software is successfully installed, we install Hadoop using a single Terminal command:

```
brew install hadoop
```

Hadoop configuration is required for correct operation. Seven files under /usr/local/Cellar/hadoop/2.7.2/libexec/etc/hadoop (the path has 2.7.2 since Hadoop 2.7.2 is installed. Other number might appear if a different version of Hadoop is installed) need modification:

- hadoop-env.sh
- mapred-env.sh
- yarn-env.sh
- core-site.xml
- hdfs-site.xml
- mapred-site.xml
- yarn-site.xml

The suggested source listings for these seven files are listed in Appendix A.

Remote login must be enabled to use Hadoop: check the Remote log-in option under "System Preferences => Sharing" first and enter the following two commands in Terminal:

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Running Hadoop involves the following commands in Terminal:

```
hadoop namenode -format
/usr/local/Cellar/hadoop/2.7.2/libexec/sbin/start-dfs.sh
```

```
/usr/local/Cellar/hadoop/2.7.2/libexec/sbin/start-yarn.
    sh
```

Note that the first command is required only for the first time after installation. To stop Hadoop, enter the commands:

```
/usr/local/Cellar/hadoop/2.7.2/libexec/sbin/stop-dfs.sh
/usr/local/Cellar/hadoop/2.7.2/libexec/sbin/stop-yarn.sh
```

From personal experience, we can attest that these commands are not required if web monitoring is not desired. If only MR is needed, Section 3.4 is all that is required after the steps described above are done once.

3.2.7 AWS Configuration

The AWS configuration includes server side and client side configurations. Although we already have an AWS account at this time, a user account is required only for CLI access and should be set up on the server side. The client side setup needs to use the information acquired during server side setup.

Server Side Configuration

Log into the AWS Console => IAM => Users => Add user. Enter the desired user name and check the "Programmatic access" option. The user should have privileges to access S3 and EMR, but administrator privilege also work. Remember to save the created access key for client side configuration.

Client Side Configuration

Follow the AWS CLI installation guide (the AWS CLI installation guide can be found at http://docs.aws.amazon.com/cli/latest/userguide/installing.html), or enter the following command in Terminal:

```
brew install awscli
```

Configure the AWS CLI by following AWS CLI configuration tutorial (the AWS CLI configuration tutorial can be found at http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html), or enter the following command in Terminal:

```
aws configure
```

Enter the Access Key information retrieved from Section 3.2.7, and enter us-east-1 for Default region name. Skip the Default output format by pressing enter without entering anything.

3.3 RWLG

The original idea for RWLG in the MR paper from Google [3] includes the following operations:

1. For each web page, record its URL as $sourceURL$.
2. For the same web page, record its link URL as $targetURL[n]$, for n different links.
3. Mapper's output would be $< targetURL, sourceURL >$ for each link.
4. Reducer gathers all Mapper's output and rearranges them by $targetURL$ as key and produces the following output value pairs: $< targetURL, list(sourceURL) >$.

3.3.1 Algorithm 3-1: Basic Algorithm

We can easily implement the basic algorithm from the original idea just described to process WARC files from the Common Crawl with the help of the Jsoup library. The algorithm is as follows with the detailed Java code source listings in Appendix B:

1. Fact: $\forall$ Warc record, $\exists$ only 1 $sourceURL$ in its header.
2. $\forall$ Warc record = HTML content, the mapper emits value pairs $< targetURL$, $sourceURL >$ for every $targetURL$ found in this specific HTML content.
3. Reducers gather the value pairs by key ($targetURL$) and generate new value pairs $< targetURL, list(sourceURL) >$ for different keys.

3.3.2 Algorithm 3-2: Algorithm for Far Links

From the result of the basic algorithm, we created an extended algorithm to find link pairs that are two hops away or more. For example, link pair (A, C) is a two-hop link to which page C has a link to an intermediate page B, and page B has a link to page A. The algorithm follows with the detailed Java code source listing in Appendix C:

1. Reverse the output from the basic algorithm back to $< targetURL, sourceURL >$ value pairs instead of lists.
2. For every $< targetURL, sourceURL >$ value pair, the mapper emits $< sourceURL, < targetURL, sourceURL >>$ and $< targetURL, < targetURL, sourceURL >>$. Since self-links cause problems when it comes to far links, we do not generate pairs for any pair that points to itself; that is, $< targetURL, sourceURL >$ where $targetURL = sourceURL$.
3. By the nature of MR, $(key, value)$ pairs with the same key are gathered by the same reducer; thus, if two web pages have the same middle URL, the received $< key, < targetURL, sourceURL >>$ has the form: $< middleURL, < targetURL, middleURL >>$ and $< middleURL, < middleURL, sourceURL >>$. We create two sets named $srcRequester$ and $srcProvider$, which stand for source requester and source provider, respectively. We then extract $targetURL$ into the $srcRequester$ set from value pairs whose key is the same as $sourceURL$ and extract $sourceURL$ into the $srcProvider$ set from value pairs whose key is the same as $targetURL$.
4. When the set is complete, the reducer generates its output value pairs $< targetURL, sourceURL >$ by iterating through every single element from both the $srcRequester$ and $srcProvider$ sets. The output represents links that are two hops away (or more if $srcRequester$ is collected from links that are more than one hop away). We want to exclude the links that point to themselves from the output in this step as well; that is, exclude $< targetURL, sourceURL >$ pairs that have $targetURL = sourceURL$.
5. Since $middleURL$ is the key, these output values are not ordered by $targetURL$ and they may be repeated in other reducers. To reorder the key and reduce redundant data, we concatenate the result with another MR job, generating value pairs $< targetURL, list(sourceURL) >$ as final output.

The illustration in Figure 3.1 helps us understand the mechanism in the first part of this algorithm. We see that a two-hop link between the target and the source is connected by the

middle. It is natural to use the middle as the key in the MR job since it participates in both sides. The mapper takes the previous link pairs as input and emits the output value pairs of form $< key, < target, source >>$. A reducer handling key=mid collects the corresponding value pairs, and the reducer easily extracts values from these pairs into either $srcRequester$ or $srcProvider$ sets. Once we complete the $srcRequester$ and $srcProvider$, we can generate the two-hop-link value pairs and sort them with the target as the key through another MR job.

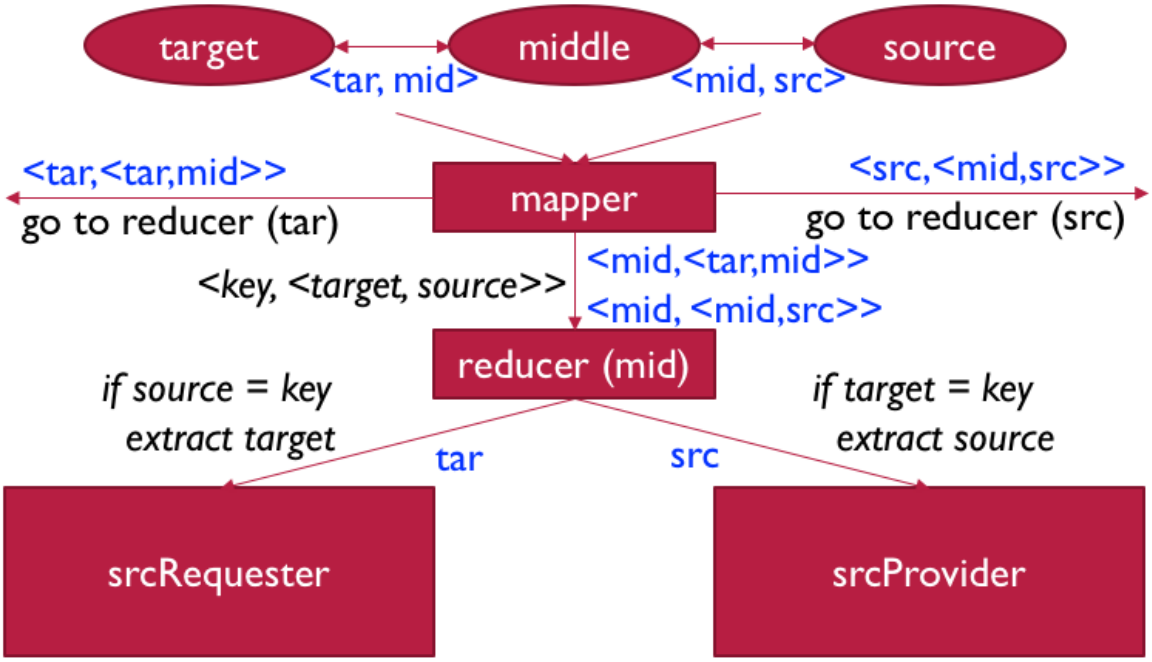


Figure 3.1. Illustrated Internal Process for Algorithm 3-2

This illustration implies that if we use n-hop link pairs on the left side and one-hop link pairs on the right side, we get $srcRequester$ and $srcProvider$ that are n+1 hops away; thus, any far links can be generated by one-hop links through this algorithm.

3.3.3 Proof and Validation for Algorithm 3-2

The algorithm for far links described in the previous section claims both that for every resulting pair $< a, c >$, the target a is always two-hops away from the source c, and for any pair $< a, c >$ that does not belong to the resulting pair, target a is never two-hops away from the source c. To prove the algorithm is working, both the analytical and experimental

approaches are provided in this section. For the analytical proof, the algorithm is proved through a direct proof for the "if" part and a proof by contradiction for the "only if" part; for the experimental validation, a record of 66 value pairs was created for testing.

Analytical Proof

Define: $b \rightarrow a \implies b$ has a link to a.

Define: $b \rightarrow\!\!\rightarrow a \implies b$ is two-hops away $sourceURL$ to a.

Define: I_1 = all $< a, b >$ value pairs from Step 2 of Algorithm 3-1.

Define: I_2 = all $< a, b >$ value pairs from Step 4 of Algorithm 3-2.

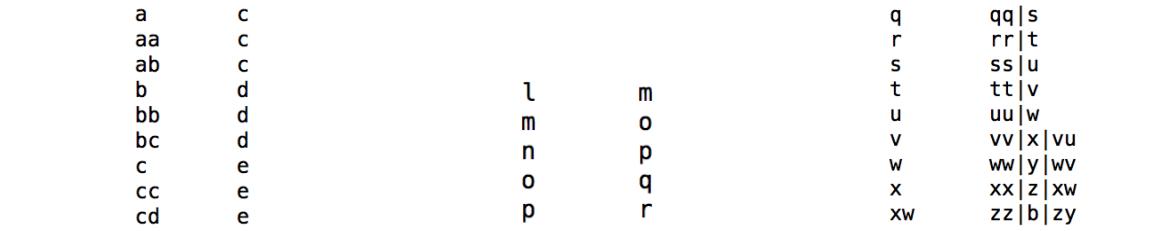
Proof: $c \rightarrow\!\!\rightarrow a \; iff \; \exists b \; s.t. \; b \rightarrow a$ and $c \rightarrow b$ and a, b, c are all distinct.

Fact: $a \neq b$ and $b \neq c$ from Step 2 of Algorithm 3-2; $a \neq c$ from Step 4 of Algorithm 3-2.

1. direct proof. $\forall < a, c > \in I_2 \implies c \rightarrow\!\!\rightarrow a$:
   $< a, c > \in I_2 \implies < a, c > \in$ Step 4 of Algorithm 3-2.
   $\implies < a, c >$ is generated from Step 2 of Algorithm 3-2.
   $\implies \exists b$ which is a $middleURL \; s.t. \; c \rightarrow b$ and $b \rightarrow a$ and a, b, c are all distinct (from Fact).
   $\implies c \rightarrow\!\!\rightarrow a$.
2. proof by contradiction. $\forall < a, c > \notin I_2 \implies c \rightarrow\!\!\rightarrow a$ is not true.
   (a) Assume $\nexists b \; s.t. \; b \rightarrow a$ and $c \rightarrow b$.
   $\implies c \rightarrow\!\!\rightarrow a$ is not true since there is no $middleURL$ between a and c.
   (b) Assume $\exists b \; s.t. \; b \rightarrow a$, $c \rightarrow b$, and $< a, c > \notin I_2$. Then $< a, b > \in I_1$ and $< b, c > \in I_1$ since I_1 has all one-hop links.
   $\implies$ Step 2 of Algorithm 3-2 generates $< b, < a, b >>$ and $< b, < b, c >>$ as output.
   $\implies$ Step 3 of Algorithm 3-2 gathers these value pairs and emits $< a, c >$ in Step 4; this means $< a, c > \in I_2$, which is a contradiction.

Experimental Validation

A file with 66 value pair records was manually created. These records include 1-to-1 pairs, 1-to-2 pairs, 2-to-1 pairs, 1-to-3 pairs, 3-to-1 pairs, and extra 1-to-1 pairs that cause duplicate paths. This test file was processed by a MR job utilizing Algorithm 3-2, and the resulting output showed the algorithm to be correct. The algorithm caught all valid pairs and discarded the replicated value pairs. The samples of the previous description are shown in Figure 3.2. Both the test input and output files are listed in Appendix D.



a	c
aa	c
ab	c
b	d
bb	d
bc	d
c	e
cc	e
cd	e

(a) Result for 1-Source-Many-Targets Pairs

l	m
m	o
n	p
o	q
p	r

(b) Result for 1-to-1 Pairs

q	qq|s
r	rr|t
s	ss|u
t	tt|v
u	uu|w
v	vv|x|vu
w	ww|y|wv
x	xx|z|xw
xw	zz|b|zy

(c) Result for Many-Sources-1-Target Pairs

Figure 3.2. Results for the Experimental Validation

3.4 Experiment Execution

How the testing and execution were done in both the local machine and AWS are described in this section. Note that these instructions work only after the steps described in Section 3.2 are correctly carried out.

3.4.1 Local Machine Execution

The steps for JAR file packaging and MR execution from a local machine are described in this section.

JAR File Packaging in Eclipse

After selecting the working project in Eclipse, navigate "File" => "Export."

Choose "Runnable JAR file" and click "Next >."

Choose the MR driver class file for "Launch configuration" and enter the desired directory for "Export destination." Select "Extract required libraries into generated JAR" for "Library handling" and click "Finish."

There usually are many driver class files in a JAR file, and setting "Launch configuration" means we cannot use other drivers. To solve this problem, open a Terminal window and navigate to the directory containing the JAR file, then enter:

```
vim [JAR file name]
:1
```

Move the cursor to the line indicating META-INF/MANIFEST.MF and press enter. Press "a" to enter insert mode, and remove the line starting with Main-Class. Press Esc and enter:

```
:wq
:q
```

The change is now saved, and we can freely choose the driver in the Hadoop command line.

MR Execution

Enter the following command under the same directory with the packaged JAR file in Terminal:

```
hadoop jar [JAR filename] [Driver class name] [input
   path] [output path]
```

For example, assume the JAR file is under /Users/Tao/MapReduce, the input file is under /Users/Tao/MapReduce/input, and the expected output directory is /Users/Tao/MapReduce/output. Move first to the same directory by

```
cd /Users/Tao/MapReduce
```

Assume the JAR filename is WebLink.jar and the Driver class name is LinkDriver and enter the following command to carry out the MR job:

```
hadoop jar WebLink.jar LinkDriver input output
```

3.4.2 AWS CLI Execution

The detailed document for the AWS CLI can always be found on the AWS website (full CLI documentation can be found in https://aws.amazon.com/cli/). The instruction here only describes commands that are used in this study.

Upload JAR File to AWS S3

To upload the JAR file, we first create a bucket using the following command:

```
aws s3api create-bucket --bucket [Bucket name]
```

Use the following command to upload the JAR file:

```
aws s3 cp [JAR filename] s3://[Bucket name]/
```

Assume the unused desired bucket name is "mapreducejob" and the JAR file name is "WebLink.jar." The command required is:

```
aws s3api create-bucket --bucket mapreducejob
aws s3 cp WebLink.jar s3://mapreducejob/
```

Delete Resulting Output Files from AWS S3

We mentioned previously that we are charged daily for the output file based on the file size; therefore, we prefer to delete any unneeded output files. The S3 GUI gets stuck when trying to remove a significant number of files, so we use the CLI to carry out the remove operation instead. The command to delete a folder and all its content in a S3 bucket is the following:

```
aws s3 rm --recursive s3://[Bucket name]/[Folder Path]/
```

Assume the bucket name is "mapreducejob" and the folder is named "output" under the mapreducejob bucket. Then, the command becomes

```
aws s3 rm --recursive s3://mapreducejob/output/
```

Execute MR job using AWS EMR

There are many options in the AWS CLI for EMR, but for this research we only need to create a cluster and add steps. The most general command used is

```
aws emr create-cluster --steps [Step options] --release-label
    [EMR version] --instance-groups [Instance options] --auto
    -terminate -- enable debugging --log-uri [Log path] --name
    [Cluster name]
```

Step options are

```
Type=CUSTOM_JAR,NAME=[Step name],ActionOnFailure=
    TERMINATE_CLUSTER,Jar=[JAR file path],Args=[Arguments]
```

where the arguments in our research are [Driver class name],[Input path],[Output path]; the up-to-date EMR version is emr-5.2.0; the instance options are:

```
InstanceGroupType=MASTER,InstanceCount=1,InstanceType=[Master
    instance type],InstanceGroupType=CORE,InstanceCount=[Core
    instance count],InstanceType=[Core instance type]
```

where the master and core instance types are listed in AWS listing (available AWS EMR instance types can be found at https://aws.amazon.com/emr/pricing/). The core instance count is the number of cores wanted in this cluster.

One typical, often used command is:

```
aws emr create-cluster --steps Type=CUSTOM_JAR,Name=
    examplestep,ActionOnFailure=TERMINATE_CLUSTER,Jar=s3://
    mapreducejob/WebLink.jar,Args=LinkDriver,s3://Common Crawl
    /crawl-data/CC-MAIN-2016-07/segments/1454701145519.33/warc
    /,s3://mapreduce/output/exampleoutput/ --release-label
    emr-5.2.0 --instance-groups InstanceGroupType=MASTER,
    InstanceCount=1,InstanceType=c3.xlarge InstanceGroupType=
    CORE,InstanceCount=19,InstanceType=c3.xlarge --auto-
    terminate --enable-debugging --log-uri 's3n://aws-logs
    -270560560283-us-east-1/elasticmapreduce/' --name '
    examplecluster'
```

CHAPTER 4:
Performance Results

We conducted the executions with our implemented algorithms using the methodology described in the previous chapter, and the results are presented in this chapter in two parts. The execution results from local execution for a single WARC file to AWS execution on the cloud for one of the latest segments in 2017 are contained in the first part. The results of a series of attempts to finish one full archive in one AWS execution are included in the second.

4.1 Experimenting with a Single Segment

In the Coudray thesis [2], success was defined based upon cloud processing of a single Common Crawl segment. We followed the same path to claim success on RWLG and delved into a larger data set from that point. We began mining a single WARC file with local execution and built towards mining a full Common Crawl segment.

4.1.1 Local Execution with One WARC File

As described in Chapter 3, the algorithms were shown to be correct through both mathematical analysis and a test case. We wanted to know how the algorithms worked when dealing with real-world data sets. Constrained by the computing power of the local machine, we used only one WARC file (CC-MAIN-20170116095119-00000-ip-10-171-10-70.ec2.internal.warc.gz) from the Common Crawl's most recent archive (CC-MAIN-2017-04) as the input.

The local executions ran on an early-2015 13-inch MacBook Air with OSX El Capitan version 10.11.16 and Hadoop version 2.7.2. The laptop was equipped with a 2.2-GHz Intel Core i7 Processor, 8-GB 1600-MHz DDR3 memory, and an Intel HD Graphics 6000 1536-MB graphic card. The results were separated by first-hop links and second-hop links, with the no-dynamic option combined and compared.

First-Hop Links

We ran the execution using the basic algorithm to obtain the first-hop links. The runs were done five times for each option and finished without error. The average resulting metrics are listed in Table 4.1. The result for the no-dynamic option finished faster and was smaller in size, which was expected.

Table 4.1. Local Execution Result Metrics for First-Hop Links

Option	Execution Time	Input Size	Output Size
Normal	6 min 4 sec	1.02 GB	831.8 MB
No-dynamic	5 min 54 sec	1.02 GB	637 MB

Second-Hop Links

We ran the execution using the algorithm for far links to obtain the second-hop links. We noted that there was no "no-dynamic" option in this algorithm. Since we used the first-hop links as input, we had no-dynamic second-hop links if the no-dynamic first-hop links were given. The runs were also done five times each and finished without error. The average resulting metrics are listed in Table 4.2.

Table 4.2. Local Execution Result Metrics for Second-Hop Links

Input	Execution Time	Input Size	Output Size
Normal	2 min 17 sec	832.2 MB	8.7 MB
No-dynamic	1 min 46 sec	637.4 MB	41.9 MB

The result with no-dynamic input finished faster because the input size was smaller. Since many dynamic URLs were treated as no-dynamic ones, more links became more relevant; thus, the output size for the no-dynamic option became larger as expected.

4.1.2 AWS Execution with One WARC File

The algorithms were shown to be working with a mathematical analysis, test case, and local testing. Because it is very expensive to own and maintain a cluster capable of processing tens of terabytes of input data, we employed AWS EMR. We wanted to know how the algorithms worked in the cloud environment using AWS EMR and compared the result with the previous local testing before we launched our job with large data sets. To see

how the same task could be carried out in AWS, we executed the algorithms on the same WARC file (CC-MAIN-20170116095119-00000-ip-10-171-10-70.ec2.internal.warc.gz) as the local execution using only one EC2 instance.

The EC2 instance used here was of the type c3.4xlarge, and the reason we chose this type of instance is described in Section 4.2.2. The c3.4xlarge instance has 16 virtual CPUs using Intel Xeon E5-2680 v2 (Ivy Bridge) Processors and was equipped with 30-GB memory and two 160-GB SSDs.

First-Hop Links

We repeated mining a single WARC file for first-hop links, but this time used cloud computing. The runs finished without error, and the average resulting metrics are listed in Table 4.3.

Table 4.3. AWS Execution Result Metrics on One WARC File for First-Hop Links with One Instance

Option	Execution Time	Input Size	Output Size
Normal	6 min	1.02 GB	825.3 MB
No-dynamic	5 min	1.02 GB	632.1 MB

The result with the no-dynamic option appeared to be 6.5 MB smaller in size than the result in local execution. Since the default number of reducers in the AWS for this run was seven, we were not able to directly compare the output files. To do so, we downloaded the first-hop-link result from AWS and ran a sorting MR job on the results both from AWS and local execution. We used the *diff* command to compare the two output files, and found only two key values in millions of records that had differences in a few bytes. We could not understand what caused the difference since we were using the same JAR to run from the identical input, but the difference should be around 10 bytes instead of 6.5 MB. We concluded the output of the AWS execution was consistent with the local execution, and the difference in size might be due to the various sizing mechanisms of different file systems. The source listing for the sorting MR job is listed in Appendix E. We also noticed that the execution time for a single c3.4xlarge instance and our local execution were similar.

Second-Hop Links

We repeated mining the first-hop results to produce the second-hop results, but this time using cloud computing. The runs finished without error, and the average resulting metrics are listed in Table 4.4.

Table 4.4. AWS Execution Result Metrics on One WARC File for Second-Hop Links with One Instance

Input	Execution Time	Input Size	Output Size
Normal	1 min	825.3 MB	8.6 MB
No-dynamic	1 min	632.1 MB	41.6 MB

We did the same comparison for the second-hop-link results as described previously, and we got a similar result after sorting. The time needed to run these AWS executions was shorter than for local execution because the c3.4xlarge instance had 16 virtual CPUs and 30-GB memory. Since we had six output files from the first-hop-link result, the c3.4xlarge instance ran more mappers at a time, achieving a faster average execution time than the local machine.

4.1.3 Execution on a Segment

We now knew that our code worked on a local machine as well as in the cloud environment. In the following sections, we explore how to process one entire archive at once, which consists of hundreds of segments and thousands of WARC files. We started executing on a segment (s3://commoncrawl/crawl-data/CC-MAIN-2017-04/) from the latest Common Crawl archive using 20 EC2 instances. We originally used m2.xlarge instances but later switched to c3.4xlarge for our work since the latter had better cost-performance. We describe the steps to determine the cost-performance between different instances in Section 4.2.2. The results are presented in the following discussion.

First-Hop Links

We executed algorithm 3-1 for first hop links on AWS with 20 c3.4xlarge instances. We only ran it once for each option. The runs finished without error, and the resulting metrics are listed in Table 4.5.

Table 4.5. AWS Execution Result Metrics on One Segment for First-Hop Links with 20 Instances

Option	Execution Time	Input Size	Output Size
Normal	31 min	593.9 GB	363.3 GB
No-dynamic	31 min	593.9 GB	258.3 GB

The execution time for both the normal and the no-dynamic option was similar to that for previous executions and remained the same for the one-segment execution. The resulting output size for the normal execution was 61.2% of the original input size, down from 79% from the result for a single WARC file. This was also found in the resulting size for the no-dynamic execution: down to 32.5% from 60.5% output size. Since different web pages could have links to the ones already processed, the ratio change in output size was expected.

Second-Hop Links

We then successfully mined the first-hop results for a full Common Crawl segment to produce the second-hop results using cloud computing; however, we were not able to finish the two-hop-link execution with the no-dynamic option after 14 tries with different configurations. The results are shown in Table 4.6.

Table 4.6. AWS Execution Result Metrics on One Segment for Second-Hop Links with 20 Instances

Input	Execution Time	Input Size	Output Size
Normal	2 hr 54 min	363.3 GB	368.8 GB
No-dynamic	failed between 2~3 hrs	258.3 GB	–

The apparently longer execution time is not investigated until Section 4.2.3. To further explore the failure issue, we reduced the input from a segment to multiple WARC files and compared the resulting size between different numbers of WARC files as shown in Table 4.7.

We could clearly see that the size of the one-hop-link output grew almost linearly as the input size increased. In comparison, the two-hop-link output grew much faster. Since we had about 560 WARC files in one segment, it was reasonable to expect more than 500 GB for the output of a segment. We know the total memory used for processing this data is 600 GB distributed among 20 c3.4xlarge instances, which also must allocate memory to the

Table 4.7. AWS Execution Output Size Comparison between Different Numbers of WARC Files with 20 Instances

Input File	First-hop-link Execution		Second-hop-link Execution	
	Execution Time	**Output Size**	**Execution Time**	**Output Size**
5 WARCs	6 min	2.8 GB	2 min	392.5 MB
10 WARCs	6 min	5.4 GB	2 min	1.1 GB
20 WARCs	7 min	10.2 GB	3 min	3.2 GB
40 WARCs	10 min	19.6 GB	6 min	9.4 GB
80 WARCs	15 min	38.1 GB	14 min	29.1 GB

operating system and background programs. On the other hand, the output size for a second-hop-link execution on one segment was already twice the size as a first-hop-link output. Since far-distanced pairs could always be generated by first-hop links, we recommend only computing it on specified pages when requested in order to save storage space.

4.2 Scaling AWS Configurations for Many Segments

From the executions in the previous sections, we established that our code worked and was capable of distributed processing and parallel computing in the cloud environment; however, as pointed out in the Coudray thesis, this type of work would encounter the "out of memory" error at some point [2]. We do not want our work to be limited by the size of the data set, as the ultimate goal is to process one entire archive at once. Herein, we describe our effort to achieve this goal, step by step.

4.2.1 Challenges

If we need to process a large data set, we do not want to manually assign the separated data sets and aggregate their results on demand. An archive in the Common Crawl is an image of the Internet at a specific time, and we desire to automatically process one full archive at once. We expected to achieve this goal by parallel computing, but we encountered an "out of memory" error.

The AWS provided various types of instances to implement our jobs, but it seemed that AWS did not provide any auto-scaling ability that could adapt to the different jobs assigned. Using the default environment setting for 20 m2.xlarge instances, we succeeded in the

execution of three 2016 Common Crawl segments with a total of about 1 TB as input at a time without error, but we could not achieve any more segments beyond that point.

We thought the AWS would provide almost infinite hardware resources (e.g., CPU, disks, and memory) upon our request, which should have been enough to run as many segments as we wanted. The truth was that, depending on the algorithm we used in MR, we encountered the "out of memory" error when processing a very large data set. We encountered the "out of memory" error until we ran the execution from four segments using 20 m2.xlarge instances. The error might also have been related to inefficient resource usage that led to poor execution efficiency, so we needed both to find a way to eliminate this error, as well as to run the execution more efficiency at the same time.

4.2.2 Initial Approach: Ad Hoc

The first approach was ad hoc. We adjusted the type and number of EC2 instances and increased the number of reducers to execute the job. This was a trial and error method. We only executed Algorithm 3-1 for this approach, and the starting point was from a failed execution on four 2016 Common Crawl segments at a 95.24% progression rate (1520/1596 tasks completed) using 20 m2.xlarge instances, with six hours and 54 minutes of execution time.

Number of Reducers

In our previous executions using m2.xlarge instances, we found that the output size was proportional to the input size, as shown in Table 4.8.

Table 4.8. Output and Input Size vs. Number of Segments

Number of Segments	Single Output File Size	Number of Output Files	Total Output Size
1	1.5 GB	130	195 GB
2	3 GB	130	390 GB
3	4.5 GB	130	585 GB

The number of output files was equal to the default number of reducers, which was closely related to the number and type of instances we were using. Since the "out of memory"

error usually occurred in reducer jobs because of Java heap space, we guessed that some variations on the number of reducers might have a good effect.

First, we tried lowering the number of the reducers from 130 to 60, but the "out of memory" error came earlier, in three hours. We then tried to increase the number of the reducers from 130 to 200, but we got the "out of memory" error again because of Java heap space after ten hours and five minutes at a 98.25% progression rate (1568/1596 tasks completed). We also got another error message: "Too [M]any fetch failures." This new error message appeared because the reducers took too much time to swap, which is known as a memory paging problem. Combined with the apparently longer execution time, we thought enlarging the number of the reducers would improve the execution progression, but if we used too many reducers, the memory paging problem caused execution failure. Considering the 3% extended progression required three more hours in execution time, the swapping takes a significant amount of time and results in fetch failure because of timeouts. Considering that we had 100 segments in an archive, increasing the number of the reducers does not solve our problem.

Types of Instances

We could always use more instances to process our work, but we wanted to use them wisely. We knew different types of instances would be suitable for different jobs, as discussed in Chapter 2. To determine the right type, we ran the basic algorithm five times on the same single segment using different types of 20 instances, and the result is shown in Table 4.9.

Table 4.9. Execution Time vs. Different Types of Instances (Same Segment)

Type	1st run	2nd run	3rd run	4th run	5th run	Avg (min)
m3.xlarge	69	66	68	60	63	65.2
r3.xlarge	62	58	61	58	59	59.6
c3.xlarge	83	73	68	68	68	72
d2.xlarge	57	57	55	57	57	56.6
i2.xlarge	61	64	60	64	66	63
g2.2xlarge	41	41	40	41	41	40.8

We could see that g2.2xlarge is the fastest of the different types of basic instances; however, we had to take the price per hour (hr) for instances into consideration. We could also

calculate the price per segment (seg) from the pricing information and the execution time, as listed in Table 4.10.

Table 4.10. Advanced Comparison between Instance Types

Type	Avg (min)	Avg (hr)	price/hr	price/seg
m3.xlarge	65.2	1.087	0.07	0.0761
r3.xlarge	59.6	0.993	0.09	0.0894
c3.xlarge	72	1.2	0.053	0.0636
d2.xlarge	56.6	0.943	0.173	0.1632
i2.xlarge	63	1.05	0.213	0.2237
g2.2xlarge	40.8	0.68	0.2	0.136

We could see that although the price/hr for g2.2xlarge was relatively high, its price/seg was quite low when the processing speed is taken into account. There are still two points to be noticed here: we considered in Chapter 2 that computation-optimized instances outperformed other types, and the price/hr for the basic m3.xlarge, r3.xlarge and c3.xlarge instances were much lower than the other three different types of basic instances. We did the runs again using the advanced instances, c3.4xlarge (four times more CPUs, memory, and disk size than c3.xlarge) and r3.2xlarge (twice the resources than r3.xlarge), which had similar price/hr to g2.2xlarge (type m3 did not have one with similar pricing), and the result is shown in Table 4.11.

Table 4.11. Advanced Comparison between Instance Types in the Same Price Range

Type	Avg (min)	Avg (hr)	price/hr	price/seg
r3.2xlarge	41.4	0.69	0.18	0.1242
c3.4xlarge	21	0.35	0.21	0.0735
d2.xlarge	56.6	0.943	0.173	0.1632
i2.xlarge	63	1.05	0.213	0.2237
g2.2xlarge	40.8	0.68	0.2	0.136

We found c3.4xlarge to be the best cost-performance instance type when doing our algorithm, and it required the least amount of time, as expected. We note that these results were only for AWS EMR pricing. The instances (ins.), which are charged by AWS EC2 pricing, should also be considered in the actual cost. The final cost results in an even greater difference as the number of instances increases, as shown in Table 4.12. Nonetheless, this

result was much better than the m2.xlarge instance we used previously, in which it took almost two hours to finish the execution from a single segment and encountered the "out of memory" error because of Java heap space after almost seven hours of execution time on four segments. We executed the algorithm again on four segments, using 20 c3.4xlarge instances, and it finished in 71 minutes without error. We decided to use c3.4xlarge as the instance type to execute our jobs since it had the fastest execution time for our job, and its price per segment was almost the lowest one, about three times faster than c3.xlarge in execution time and only 15% higher in price/seg for EMR.

Table 4.12. Combined Cost Comparison between Instance Types

Type	Avg (hr)	EMR pr/hr	EC2 pr/hr	20 ins. pr/seg	40 ins. pr/seg	60 ins. pr/seg	80 ins. pr/seg	100 ins. pr/seg
r3.2xlarge	0.69	0.18	0.665	9.3	18.48	27.66	36.83	46.01
c3.4xlarge	0.35	0.21	0.84	5.95	11.83	17.71	23.59	29.47
d2.xlarge	0.943	0.173	0.69	13.18	26.19	39.20	52.22	65.23
i2.xlarge	1.05	0.213	0.853	18.14	36.05	53.96	71.88	89.79
g2.2xlarge	0.68	0.2	0.65	8.98	17.82	26.66	35.5	44.34

Number of Instances

As soon as we succeeded on four segments using 20 c3.4xlarge instances, we tried the full archive using 100 c3.4xlarge instances, and we failed. After 12 hours, we received the "out of memory" error because of both "Java heap space" and "GC overhead limit exceeded." We then reduced the number of segments to 18, a number close to 20 and easily assigned because of the naming convention in the Common Crawl, to try again since we already knew that 20 instances could finish four segments at once. If we were doing parallel computing, then 100 instances should have been more than enough for a fluent execution from 18 segments. After 82 minutes of execution, the "out of memory" error appeared again with a 98.77% progression rate (6979/7066 task completed), and we realized that we were not able to solve this error by simply increasing the number of instances.

Conclusion for Ad Hoc Approach

The most beneficial information we learned from the ad hoc approach was that c3.4xlarge is the best choice over different instance types for both cost-performance and speed in this study. We also learned that slightly increasing the number of the reducers can possibly

overcome the "out of memory" error if the task progression is high enough before the error occurs. Furthermore, we discovered that increasing the number of instances was not likely to solve the "out of memory" problem caused by "Java heap space." We realized the need to dive deeper into the environment settings in HMR in order to achieve the goal.

4.2.3 Systematic Approach

After the ad hoc approach, we realized that the AWS does not provide auto-scaling when we use more instances, and we adjusted the environment variables, especially for the memory. We then revisited the YARN architecture and tried to understand how to adjust the environmental variables. Fortunately, we discovered a rule of thumb with which to begin [8]. We also looked into the previous error message and sorted out a list of possible variables related to various types of errors. The different types of errors, the corresponding variables that might remove the errors, and the related files in which each variable resided are listed in Table 4.13. In this approach, we adjusted each variable in order to understand how they affect the results.

Table 4.13. Types of Error and Possible Corresponding Environmental Variables

Error Type	Corresponding Variable	Related File
Java heap space	yarn.nodemanager.resource.memory-mb (1) yarn.scheduler.minimum-allocation-mb (2) yarn.scheduler.maximum-allocation-mb (3) mapreduce.task.io.sort.mb mapreduce.map.memory.mb (4) mapreduce.reduce.memory.mb (5) mapreduce.map.java.opts (6) mapreduce.reduce.java.opts (7) yarn.app.mapreduce.am.resource.mb (8) yarn.app.mapreduce.am.command-opts (9)	yarn-site.xml mapred-site.xml
Too many fetch failures	mapreduce.job.reduce.slowstart.completedmaps mapreduce.shuffle.max.threads mapreduce.reduce.shuffle.parallelcopies	mapred-site.xml
GC overhead limit exceeded	HADOOP_DATANODE_OPTS HADOOP_CLIENT_OPTS	hadoop-env.sh

To change the environment variable through AWS CLI, we created a configuration file and

referred to the file named "configurations.json" when creating the cluster with the following argument:

```
--configurations file://configurations.json
```

The configuration file had a certain format, which is described in Appendix F.

Stage 1: 18 Segments

We failed while executing from the input of 18 segments, so we used 18 segments as the starting point for this approach. Since the "out of memory" error was our major concern, we focused on the memory-related parameter adjustments. We concluded four different settings toward successful execution, and the variable set used is shown in Table 4.14. Because of the long variable name, we indicated the variable using the number assigned in Table 4.13.

Table 4.14. Variable Sets and Results for 18 Segments

Item	**1**	**2**	**3**	**4**
# of instances	20	20	21	21
var (1)	20480	614400	614400	35840
var (2)	4096	8500	8500	7168
var (3)	20480	614400	614400	35480
var (4)	4096	8500	8500	7168
var (5)	8192	17000	17000	14336
var (6)	-Xmx3276m	-Xmx6800m	-Xmx6800m	-Xmx5734m
var (7)	-Xmx6553m	-Xmx14600m	-Xmx14600m	-Xmx11469m
var (8)	8192	17000	17000	14336
var (9)	-Xmx6553m	-Xmx14600m	-Xmx14600m	-Xmx11469m
execution time	11hr29min	9 min	5 min	11hr5min
progression	99.67%	0%	0%	100%
error code	143	137	137	–

We found a post on the Internet that also ran a comparably large MR job and got the "out of memory" error [29]. The originator of the post finally succeeded using a set of environment variables and shared this information freely. We used this setting with only 20 instances to run the first execution, which took about 11.5 hours before the "out of memory" error occurred and reached a 99.67% progression rate, which is almost 1 percent better than the previous execution using 100 instances with the default setting. This proved adjusting memory-related parameters helped to improve the outcome of the execution.

Unfortunately, there is only a little information as to how to set these numbers. Most guidance found in AWS manuals and on the Internet, which were reasonable but useless, indicated that these variables were important and needed to be adjusted depending on the job we were running. The only reference we found described a general rule to calculate the recommended memory setting depending on system capacities [8]. We followed the instruction to set the environment variables for the second execution. It failed because the system could not allocate the assigned size of memory, which we thought was because of a feature provided by AWS EMR. As stated in Chapter 2, EMR is a PaaS, which hides many system details from the user. In the second execution, one of our 20 instances was assigned as a "master" instance. It only ran the ResouceManager application but neither map nor reduce jobs. For the third execution, we increased the requested number of instances to 21, so that we would have 20 instances running mappers and reducers.

The third execution also failed due to memory allocation. We realized that there were some inconsistencies between [8] and EMR. We did not investigate this issue at this stage, but we thought the unrealistic memory variables var (1) and var (3) might be the problem. We tried to lower these numbers and finally succeeded using the fourth setting, with about 11 hours of execution time.

At this stage, we established the starting point to properly set up the AWS EMR environment for the jobs to successfully execute a much larger job than the previous executions. We learned that error code 143 was for the "out of memory" issue, and error code 137 was related to the memory configuration error. We achieved executing 18 segments as input. This was six times larger than three segments as input, which was our original capability using 20 m2.xlarge instances.

Stage 2: 34 Segments

After our success with 18 segments, we used the same setting to run the full archive and failed. We knew that we did not fully understand the memory settings, such as the inconsistencies between [8] and EMR, and we needed more adjustments to gain a better understanding of the memory settings. We reduced the input size to 34 segments and executed with 41 instances, where one of them was assigned as the "master" using the successful setting in 18 segments and again got the "out of memory" error. We ran different executions, which yielded the eight results captured in Table 4.15 and Table 4.16. As on

the previous stage, the variables are indicated by the number assigned in Table 4.13.

Table 4.15. Variable Sets and Results for 34 Segments (part 1)

Item	**1**	**2**	**3**	**4**
# of instances	41	41	41	41
var (1)	70656	35328	70656	25500
var (2)	7065	7065	7065	7065
var (3)	35328	35328	70656	25500
var (4)	7065	7065	7065	–
var (5)	14131	14131	21504	14500
var (6)	-Xmx5652m	-Xmx5652m	-Xmx5652m	–
var (7)	-Xmx11305m	-Xmx11305m	-Xmx17203m	-Xmx11500m
var (8)	14131	14131	21504	14500
var (9)	-Xmx11305m	-Xmx11305m	-Xmx17203m	-Xmx11500m
execution time	4hr24min	12hrs	6hr17min	6hr33min
progression	70.34%	99.95%	97.23%	99.95%
encountered error	137	143	137	143

Table 4.16. Variable Sets and Results for 34 Segments (part 2)

Item	**5**	**6**	**7**	**8**
# of instances	41	41	45	41
var (1)	25500	25500	25500	23000
var (2)	250	250	250	250
var (3)	25500	25500	25500	23000
var (5)	22500	14000	14131	16500
var (7)	-Xmx18000m	-Xmx11000m	-Xmx11305m	-Xmx13500m
var (8)	22500	14000	14131	16500
var (9)	-Xmx18000m	-Xmx11000m	-Xmx11305m	-Xmx13500m
execution time	7hr56min	6hr31min	6hr7min	6hr23min
progression	70.34%	97.23%	99.95%	100%
encountered error	143	143	143	–

We started by determining the proper setting for variables var (1) and var (3) with different settings between set 1 and 3. We also compared the relations between the element components of YARN, the results from AWS EMR, and the reason why variables were set as in [8].

After a series of executions, it became clear that the settings in [8] are for a cluster of a single instance. This explains why variables var (1) and var (3) add up to the total memory in a single instance in [8]. In EMR, every core instance was a single computer running both the DataNode and NodeManager daemons. Since we used c3.4xlarge instances as core instances with the default setting information given in [30], [31], the maximum total memory was around 27 GB per instance and should be the number for variables (1) and (3). We also assumed that the size of each container was decided by variables (2) and (3), which was first set by variable (2) and was increased upon request by multiples of variable (2) until it reached the number indicated in variable (3).

Reference [8] indicates that the reducer memory allocation, var (5), should be twice the mapper memory allocation, var (4). Furthermore, var (2) is the minimum container memory allocation and should be the greater common divisor of var (4) and var (5). We also found that we did not have to set a larger number for variable (4) because the default value for the mapper is always larger than needed, and we never encountered errors in mapper tasks. Actually, we removed variables (4) and (6) and achieved a much faster execution time, which saved more than five hours for the same progression rate. This was because we can have many more mappers running at the same time. Tracking the causes of errors became easier once we knew what to adjust by the given error code.

Since it increased the execution time, we removed the modification on mapper memory parameters in sets 5 through 7. We slightly adjusted the reducer memory in sets 5 and 6 using set 4 as a template and tried increasing the number of instances in set 7, but all executions for variable sets 5, 6, and 7 failed.

We noted that the progression for set 5 was surprisingly low, so we looked into the error message in each failing task. The general error message output from the cluster console did not give any useful information, so we had to track down the error to each assigned task and each container. We found that they failed not because of the "out of memory" error as given in the cluster console. Instead, they failed because the reducer task had issues connecting to S3 when emitting the result, which happened rarely when many users were accessing S3 at the same time. We decided to use a larger reducer memory setting to execute, as listed in set 8, for which the amount was decided by observing the trends of progression and reducer memory. We finally succeed after about 6.5 hours.

At this stage, we learned the meaning of most of these memory-related variables and the roles they played in the YARN model. We also learned how to track down the source of an error to find the actual reason for the problem, instead of blindly believing the given error message. Hints for tracking an error message are shown in Appendix G. We also realized that as a PaaS, AWS EMR had some downsides. EMR depended on S3 to save the output and did not keep the cluster once the job was terminated. This meant our effort was lost once there was a connection inconsistency between EMR and S3.

Stage 3: Full Archive

From the previous stages, we thought we learned enough about YARN, and we started the execution using one full 2016 Common Crawl archive (CC-MAIN-2016-07) as the input on 100 c3.4xlarge instances. The results are shown in Table 4.17.

Table 4.17. Variable Sets and Results for the Full Archive

Item	1	2	3
# of instances	100	100	100
yarn.nodemanager.resource.memory-mb	27468	27468	27468
yarn.scheduler.minimum-allocation-mb	1024	160	32
yarn.scheduler.maximum-allocation-mb	27468	27468	27468
mapreduce.map.memory.mb	1024	–	–
mapreduce.reduce.memory.mb	20480	20480	22528
mapreduce.map.java.opts	-Xmx864m	–	–
mapreduce.reduce.java.opts	-Xmx16384m	-Xmx16384m	-Xmx18022m
yarn.app.mapreduce.am.resource.mb	20480	20480	22528
yarn.app.mapreduce.am.command-opts	-Xmx16384m	-Xmx16384m	-Xmx18022m
execution time	7hr8min	7hr37min	7hr51min
progression	99.99%	99.99%	99.99%
encountered error	143	143	143

From previous executions, we thought the mapper would suffice with just 1-GB memory, which was the default for m2.xlarge for faster execution speed. We set the mapper memory to be lower than the default setting because if the job finished earlier, the price would be much lower. We increased the reducer memory because the ratio between number of instances and number of segments had been decreased to one. The resulting set 1 failed at a 99.99% progress rate since the assigned 1-GB memory limitation caused two out of 35684 assigned

tasks to fail, with one task cancelled because of the failed tasks. We fixed the problem by removing the map memory settings as in the variable set 2, but we again encountered the "out of memory" error from the reducers. We increased the reducer memory settings as in the variable set 3. We still had the "out of memory" error, which indicated that we still did not allocate enough memory. We knew that we could succeed at the price of a longer execution time if we assigned more memory to the reducer because the number of mappers running at the same time would be reduced. Because we had some administration issues to run more executions on the AWS and the 99.99% complete MR job had generated usable output with 18.3 TB in size, we did not continue with further executions.

Conclusion for Systematic Approach

We learned a lot with the systematic approach. We now clearly understood the YARN architecture and how it was affected by the environment variables. We learned how to track down the error message to the source of the problem in either a mapper/reducer task or a container. We also learned the limits of EMR as a PaaS. Most importantly, we essentially achieved our original goal: to complete a full archive in a single execution.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions and Recommendations

Two goals of the study described in this thesis support the larger NPS research project: the first goal is to implement RWLG, a search capability for web sites, and the second goal is to take a full Common Crawl archive as the input to be processed in one MR job. We essentially achieved both goals successfully in this study, as we had an almost successful execution, we had generated 99.9% of usable result in the second goal, and we know exactly how to adjust the settings to complete the execution; however, there are still possible avenues to be explored, such as implementing other algorithms or making use of the output from this work. We summarize our contributions and provide recommendations for future work in this chapter.

5.1 Conclusions

The implementation of RWLG was accomplished successfully, and the algorithm was almost successfully executed on a full 2016 Common Crawl archive. We also extended the RWLG algorithm to be capable of generating link pairs that are more than one hop away, with both analytical proof and experimental validation provided. Using [2] as a building block, we shared most of the same tools in this study. Other than providing the powerful RWLG as a search capability, MR programs that can categorize, sort, and compare the stored value pairs to utilize the resulting output were also discussed. By working with low-level, relatively stable CLI commands in this study, we were able to run these programs on AWS independently from its GUI interface, which may change from time to time.

The existing YARN has been successfully leveraged to better support the extremely large data sets processed for this thesis research. Through the systematic approach, we achieved the goal of taking a full Common Crawl archive as input to be processed in a single MR job by identifying and setting the suitable environment variables for HMR components. Providing the insight of how to track down an error to the original cause, we can use this thesis to serve as a guide to adjust the corresponding variables to achieve a faster execution time while successfully finishing different MR jobs.

5.2 Recommendations

Although we essentially achieved both goals, there are still many possibilities for further research. The possible areas to explore are listed below.

1. Design and implement new search capabilities
   Previous work provided the inverted index, and this work provides RWLG. There are still different search capabilities to be designed and implemented to help find the desired information efficiently. Other than designing a new search capability, another path for future work can combine the results of the inverted index and RWLG to offer more precise information for any preferred application.
2. Move to IaaS instead of PaaS
   As stated in Chapter 4, there are downsides for PaaS. Not only did we lose data once an error occurred, we noted that low level system details are hidden from the user. If we switch to EC2 to use a set of instances that are fully under our control, we would be able to eliminate this problem. Further, we are only allowed to use the same type of instances for deploying mappers and reducers. This is no longer true if we switch to EC2, in which we can freely choose different types of instances that fit our needs and there are more kinds of instances from which to choose. In IaaS, we assume the primary responsibility is in configuring the operating system, Hadoop, and the associated labor.
3. Make use of the result of this study
   We now have the RWLG output of a full archive and want to make use of it. One way to use these value pairs is to write a search program that takes an operator's input and uses the RWLG data produced in this work to return a list of source URLs that link to the input URL(s) in either text or graphic form. The program can also take a number n as a parameter to produce the 1-to-n-hop link graph centered on the input URL.
4. Process, analyze and compare the output of this study
   We are now capable of processing a full archive at a time, so we can analyze the different outputs using different archives. We can observe the numbers of the inward and outward links of some popular websites and compare our result with other companies' such as Alexa or Wayback Machine (Alexa, which can be found at http://www.alexa.com/, is a company from amazon.com and keeps track of traffics, statistics and analyses of different websites; Wayback Machine, which can be found

at http://archive.org/web/, is a service from Internet Archive that has saved different versions of web pages for 20 years [32]). Although these websites do not generate RWLGs as was done in this thesis, we can still determine if these companies' results are related to ours by observing the deviation in links from and to different well-known websites over the past decades.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Hadoop Configuration Files

The suggested source listings for the seven files that are required for Hadoop configuration are provided in this appendix. These seven files should be under /usr/local/Cellar/hadoop/[Version #]/libexec/etc/hadoop if they exist. Create one if needed.

A.1 hadoop-env.sh

Append the following three lines to the end of the file:

```
export JAVA_HOME="$(/usr/libexec/java_home)"
export HADOOP_OPTS="${HADOOP_OPTS} -Djava.security.krb5.
    realm= -Djava.security.krb5.kdc="
export HADOOP_OPTS="${HADOOP_OPTS} -Djava.security.krb5.
    conf=/dev/null"
```

A.2 mapred-env.sh

Append the following line to the end of the file:

```
export JAVA_HOME="$(/usr/libexec/java_home)"
```

A.3 yarn-env.sh

Append the following line to the end of the file:

```
ARN_OPTS="$YARN_OPTS -Djava.security.krb5.realm=OX.AC.UK -
    Djava.security.krb5.kdc=kdc0.ox.ac.uk:kdc1.ox.ac.uk"
```

A.4 core-site.xml

Rewrite the file content as the following code:

```
<configuration>
 <property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-${user.name}</value>
  <description>Temporary base directories.</description>
 </property>
 <property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:9000</value>
 </property>
</configuration>
```

A.5 hdfs-site.xml

Rewrite the file content as the following code:

```
<configuration>
 <property>
 <name>dfs.replication</name>
 <value>1 </value>
 </property>
</configuration>
```

A.6 mapred-site.xml

Rewrite the file content as the following code:

```
<configuration>
 <property>
  <name>mapred.job.tracker</name>
  <value>localhost:9001</value>
 </property>
</configuration>
```

A.7 yarn-site.xml

Rewrite the file content as the following code:

```
<configuration>
 <property>
  <name>yarn.resourcemanager.resourcetracker.address</name>
  <value>$resourcemanager.full.hostname:8025</value>
  <description>Enter your ResourceManager hostname.</
     description>
 </property>
 <property>
  <name>yarn.resourcemanager.scheduler.address</name>
  <value>$resourcemanager.full.hostname:8030</value>
  <description>Enter your ResourceManager hostname.</
     description>
 </property>
 <property>
  <name>yarn.resourcemanager.address</name>
  <value>$resourcemanager.full.hostname:8050</value>
  <description>Enter your ResourceManager hostname.</
     description>
 </property>
 <property>
  <name>yarn.resourcemanager.admin.address</name>
  <value>$resourcemanager.full.hostname:8041</value>
  <description>Enter your ResourceManager hostname.</
     description>
 </property>
 <property>
  <name>yarn.nodemanager.local-dirs</name>
  <value>/grid/hadoop/hdfs/yarn,/grid1/hadoop/hdfs/yarn</
     value>
  <description>Comma separated list of paths. Use the list
     of directories from $YARN_LOCAL_DIR.
```

```
    For example, /grid/hadoop/hdfs/yarn,/grid1/hadoop/hdfs/
       yarn.</description>
  </property>
  <property>
   <name>yarn.nodemanager.log-dirs</name>
   <value>/var/log/hadoop/yarn</value>
   <description>Use the list of directories from $
      YARN_LOG_DIR.
    For example, /var/log/hadoop/yarn.</description>
  </property>
</configuration>
```

APPENDIX B:
Basic Algorithm Source Listings

Two sections are contained in this appendix. In the first section, the basic algorithm source listings described in Chapter 3 are provided, and the MR job source listings used to categorize the output value pairs are given in the second. Normally, the output is ordered by the keys in alphabetical order throughout all output files. When we want to look for a specific record, we have no idea where it is. The "categorizer" MR job first extracts the target's domain name and then uses the Java built-in hashCode() function to get the hash code corresponding to the domain name. The reducer gathers these value pairs by their hash codes and writes these value pairs into corresponding files for which we can easily find the record by providing the target's domain name.

B.1 Basic Algorithm Source Listings

The MR code always has three parts: driver, mapper and reducer. Coudray's thesis grouped them into two separated files, but I prefer to put each of them into different files. These three files are listed in the following sections.

B.1.1 Driver (RWLGHTMLDriver.java)

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import nl.surfsara.warcutils.*;

public class RWLGHTMLDriver extends Configured implements Tool{
  public static void main(String[] args) throws Exception{
```

```
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf, new RWLGHTMLDriver(),
        args));
  }

  @Override
  public int run(String[] arg0) throws Exception {
    Configuration conf = this.getConf();

    Job job = Job.getInstance(conf, "RIGHTMLDriver");

    // setup all classes
    job.setJarByClass(RWLGHTMLDriver.class);
    if(arg0.length>2 && arg0[2].equals("no-dynamic")){
      job.setMapperClass(RWLGHTMLMapperND.class);
    }
    else{
      job.setMapperClass(RWLGHTMLMapper.class);
    }
    job.setReducerClass(RWLGHTMLReducer.class);

    // input/output setup
    job.setInputFormatClass(WarcInputFormat.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    // path setup
    String inputPath = arg0[0];
    String outputPath = arg0[1];

    FileInputFormat.addInputPaths(job, inputPath);
    FileOutputFormat.setOutputPath(job, new Path(outputPath));
    return job.waitForCompletion(true) ? 0 : 1;
  }
```

```
}
```

B.1.2 Mapper (RWLGHTMLMapper.java)

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashSet;
import java.util.Set;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import org.jwat.warc.WarcRecord;
import org.jwat.common.Payload;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;

public class RWLGHTMLMapper extends Mapper<LongWritable,
    WarcRecord, Text, Text>{

  // Read from WARC file
  // output value pairs <target, source1>, <target, source2>, ...
  // # Step 2, Algorithm 3-1
  @Override
  public void map(LongWritable key, WarcRecord value, Context
      context) throws IOException, InterruptedException{
    // only check the http response
    if(value.header.warcTypeStr.equals("response")){
      Payload payload=value.getPayload();
      if(payload != null){
        // get the HTML page into contStream => warcContent
        InputStream inStream = payload.getInputStream();
```

```
      ByteArrayOutputStream contStream = new
          ByteArrayOutputStream();

      // a trick to parse the warc record into String
      byte[] buf = new byte[1024];
      int len;
      while((len = inStream.read(buf)) != -1)
          contStream.write(buf, 0, len);
      String warcContent = contStream.toString("UTF-8").trim();

      if(warcContent != null && !warcContent.isEmpty()){
        // parse the HTML into objects
        Document doc = Jsoup.parse(warcContent,
            value.header.warcTargetUriStr);
        Text source = new
            Text(value.header.warcTargetUriStr.trim());
        // check all links and combine with the URL of current
            page
        Set<Text> linkSet = new HashSet<Text>();
        for(Element e: doc.select("a[href]")){
          // clear out characters that should not appear in
              the URL field
          String href =
              e.attr("abs:href").replaceAll("[\\t\\n\\r\\s]*",
              "");
          if(!href.isEmpty()){
            Text target = new Text(href);
            // outputs the non-repeated value pairs
            if(linkSet.add(target)) context.write(target,
                source);
          }
        }
      }
    }
}
```

```
  }
}
```

B.1.3 Mapper for Non-Dynamic URLs (RWLGHTMLMapperND.java)

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashSet;
import java.util.Set;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import org.jwat.warc.WarcRecord;
import org.jwat.common.Payload;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;

// This mapper discards the dynamic part of URLs
public class RWLGHTMLMapperND extends Mapper<LongWritable,
    WarcRecord, Text, Text>{

  // Read from WARC file
  // output value pairs <target, source1>, <target, source2>, ...
  // # Step 2, Algorithm 3-1
  @Override
  public void map(LongWritable key, WarcRecord value, Context
      context) throws IOException, InterruptedException{
    if(value.header.warcTypeStr.equals("response")){
      Payload payload=value.getPayload();
      if(payload != null){
        InputStream inStream = payload.getInputStream();
```

```
ByteArrayOutputStream contStream = new
    ByteArrayOutputStream();
byte[] buf = new byte[1024];
int len;
while((len = inStream.read(buf)) != -1)
    contStream.write(buf, 0, len);
String warcContent = contStream.toString("UTF-8").trim();

if(warcContent != null && !warcContent.isEmpty()){
  String s = value.header.warcTargetUriStr.trim();
  Document doc = Jsoup.parse(warcContent, s);

  // remove dynamic part in source
  if(s.contains("?")){
    String parts[] = s.split("\\?");
    s = parts[0];
  }

  Text source = new Text(s);
  Set<Text> linkSet = new HashSet<Text>();

  for(Element e: doc.select("a[href]")){
    String href =
       e.attr("abs:href").replaceAll("[\\t\\n\\r\\s]*",
       "");
    if(!href.isEmpty()){
      // remove dynamic part in target
      if(href.contains("?")){
        String parts[] = href.split("\\?");
        href = parts[0];
      }
      Text target = new Text(href);
      if(linkSet.add(target)) context.write(target,
          source);
    }
```

```
            }
          }
        }
      }
    }
}
```

B.1.4 Reducer (RWLGHTMLReducer)

```
import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class RWLGHTMLReducer extends Reducer<Text, Text, Text,
    Text>{

  // Read value pairs <target, source1>, <target, source2>, ...
  // output value pairs <target, list(source1), (target,
     source2), ...
  // Step 3, Algorithm 3-1
  // also serve as step 5 reducer for Algorithm 3-2
  @Override
  public void reduce(Text key, Iterable<Text> values, Context
     context) throws IOException, InterruptedException {
    Iterator<Text> it = values.iterator();
    Set<String> sources = new HashSet<String>();

    // clean up repeated and null/empties
    while(it.hasNext())  sources.add(it.next().toString());
    sources.remove(Arrays.asList("", null));
```

```
    Iterator<String> it2 = sources.iterator();
    StringBuilder strBdr = new StringBuilder();

    // append the first value
    strBdr.append(it2.next());
    // append every other "| value" if there's any
    while(it2.hasNext()) strBdr.append("|").append(it2.next());

    // outputs the value pair
    context.write(key, new Text(strBdr.toString()));
  }
}
```

B.2 Categorizer Source Listings

These MR codes categorize the output files into different file names by the hashed value of their keys. As described before, these codes are separated in three different files as well.

B.2.1 Driver (CategorizerDriver.java)

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class CategorizerDriver extends Configured implements
    Tool{
  public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf, new CategorizerDriver(),
        args));
```

```
  }

  @Override
  public int run(String[] arg0) throws Exception {
    Configuration conf = this.getConf();

    Job job = Job.getInstance(conf, "CategorizerDriver");

    // setup all classes
    job.setJarByClass(CategorizerDriver.class);
    job.setMapperClass(CategorizerMapper.class);
    job.setReducerClass(CategorizerReducer.class);

    // input/output setup
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    // path setup
    String inputPath = arg0[0];
    String outputPath = arg0[1];

    FileInputFormat.addInputPaths(job, inputPath);
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    return job.waitForCompletion(true) ? 0 : 1;
  }
}
```

B.2.2 Mapper (CategorizerMapper.java)

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
```

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class CategorizerMapper extends Mapper<LongWritable,
   Text, Text, Text>{
  // adjust the number of divisor (x) to get corresponding
     number of files (2x) per reducer
  private int space = (int) (((long) Integer.MAX_VALUE -
     (long) Integer.MIN_VALUE)/10);

  // Read from format: target <tab> list(source)
  // to (hashCode(target domain), target <tab>
     list(source)), ...
  public void map(LongWritable key, Text value, Context
     context) throws IOException, InterruptedException{
    String[] token = value.toString().split("\t");
    try{
      context.write(new Text(Integer.toString(new
         URL(token[0].toString()).getHost().hashCode()/space)),
         value);
    } catch(MalformedURLException ex){
      context.write(new
         Text(Integer.toString("MalformedURL".hashCode()/space)),
         value);
    }
  }
}
```

B.2.3 Reducer (CategorizerReducer.java)

```
import java.io.IOException;
import java.util.Iterator;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import
   org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;

public class CategorizerReducer extends Reducer<Text, Text,
   Text, Text>{
  private MultipleOutputs<Text, Text> mos;

  public void setup(Context context) throws IOException,
     InterruptedException{
    mos = new MultipleOutputs<Text, Text>(context);
  }

  // input: <HashCode(target domain), <target <tab>
     list(sources)>>
  // output: <target, list(sources)> into files named by
     the hash code
  @Override
  public void reduce(Text key, Iterable<Text> values,
     Context context) throws IOException,
     InterruptedException {
    Iterator<Text> it = values.iterator();
    while(it.hasNext()){
      String[] token = it.next().toString().split("\t");
      mos.write(new Text(token[0]), new Text(token[1]),
         key.toString());
    }
  }

  public void cleanup(Context context) throws IOException,
     InterruptedException{
    mos.close();
```

```
    }
}
```

APPENDIX C:
Algorithm for Far Links Source Listings

The algorithm for far links source listings is provided in this appendix. As described in Chapter 3, this algorithm concatenates two MR jobs, so this source file contains two mappers, two reducers, and a driver that defines two concatenated MR jobs. Since the result of the first MR job is temporarily saved in the cluster's disk space, this program requires a lot of disk space to execute. We suggest that one only find the far link pairs for a fixed number of pairs. If the overall far link pairs are required, consider writing another driver to separate this program into two different ones and execute them separately.

C.1 Driver (L2Driver.java)

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class L2Driver extends Configured implements Tool{
  public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf, new L2Driver(), args));
  }

  @Override
  public int run(String[] arg0) throws Exception {
    Configuration conf = this.getConf();
    FileSystem fs = FileSystem.get(conf);
```

```
// job1
Job job = Job.getInstance(conf, "L2Driver");

// setup all classes
job.setJarByClass(L2Driver.class);
job.setMapperClass(L2Mapper.class);
job.setReducerClass(L2Reducer.class);

// input/output setup
job.setOutputKeyClass(Text.class);

// path setup
String inputPath = arg0[0];
String outputPath = arg0[1];
String tempPath = "tempComp";

if(fs.exists(new Path(tempPath)))
  fs.delete(new Path(tempPath), true);
FileInputFormat.addInputPaths(job, inputPath);
FileOutputFormat.setOutputPath(job, new Path(tempPath));

job.waitForCompletion(true);

// job2
Job job2 = Job.getInstance(conf, "L2Compact");

// setup all classes
job2.setJarByClass(L2Driver.class);
job2.setMapperClass(L2CompactMapper.class);
job2.setReducerClass(RWLGHTMLReducer.class);

job2.setMapOutputKeyClass(Text.class);

FileInputFormat.addInputPath(job2, new Path(tempPath));
```

```
    FileOutputFormat.setOutputPath(job2, new Path(outputPath));

    return job2.waitForCompletion(true) ? 0 : 1;
  }
}
```

C.2 First Mapper (L2Mapper.java)

```
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class L2Mapper extends Mapper<LongWritable, Text, Text,
   Text>{

  // input: target <tab> source1|source2|source3...
  // output: (target, (target, source1)), (source1, (target,
     source1)), ...
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException{
    String[] token = value.toString().split("\t");
    String[] src = token[1].split("\\|");
    // Step 1, Algorithm 3-2
    for(String s : src){
      // Step 2, Algorithm 3-2
      if(s.equals(token[0]));
      else{
        // srcRequester
        context.write(new Text(s), new Text(token[0]+"\t"+s));
        // srcProvider
        context.write(new Text(token[0]), new
           Text(token[0]+"\t"+s));
      }
```

```
    }
  }
}
```

C.3 First Reducer (L2Reducer.java)

```
import java.io.IOException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

// input: srcRequester & srcProvider
// output: unordered <target, source> pairs that are 2-hops away
// Step 3, Algorithm 3-2
public class L2Reducer extends Reducer<Text, Text, Text, Text>{
  @Override
  public void reduce(Text key, Iterable<Text> values, Context
     context) throws IOException, InterruptedException {
    Iterator<Text> it = values.iterator();

    Set<String> srcProvider = new HashSet<String>();
    Set<String> srcRequester = new HashSet<String>();
    while(it.hasNext()) {
      String[] tarsrc = it.next().toString().split("\t");
      // check if srcProvider
      if(tarsrc[0].equals(key.toString())) // should add these
         src to local list
        // add into list of sources
        srcProvider.add(tarsrc[1]);
      else // these are the ones request the list
        srcRequester.add(tarsrc[0]);
    }
```

```
    // Step 4, Algorithm 3-2
    // output the unordered results
    for(String s: srcProvider)
      for(String t: srcRequester)
        if(!s.equals(t)) context.write(new Text(t), new Text(s));
  }
}
```

C.4 Second Mapper (L2CompactMapper.java)

```
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class L2CompactMapper extends Mapper<LongWritable, Text,
   Text, Text>{

  // Read from format: target <tab> source
  // to (target, source), ...
  // Step 6 mapper, Algorithm 3-2
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException{
    String[] token = value.toString().split("\t");
    context.write(new Text(token[0]), new Text(token[1]));
  }
}
```

C.5 Second Reducer (RWLGHTMLReducer.java)

```
import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
```

```
import java.util.Iterator;
import java.util.Set;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class RWLGHTMLReducer extends Reducer<Text, Text, Text,
    Text>{

  // Read value pairs <target, source1>, <target, source2>, ...
  // output value pairs <target, list(source1), (target,
     source2), ...
  // Step 3, Algorithm 3-1
  // also serve as step 5 reducer for Algorithm 3-2
  @Override
  public void reduce(Text key, Iterable<Text> values, Context
     context) throws IOException, InterruptedException {
    Iterator<Text> it = values.iterator();
    Set<String> sources = new HashSet<String>();

    // clean up repeated and null/empties
    while(it.hasNext())  sources.add(it.next().toString());
    sources.remove(Arrays.asList("", null));

    Iterator<String> it2 = sources.iterator();
    StringBuilder strBdr = new StringBuilder();

    // append the first value
    strBdr.append(it2.next());
    // append every other "| value" if there's any
    while(it2.hasNext()) strBdr.append("|").append(it2.next());

    // outputs the value pair
    context.write(key, new Text(strBdr.toString()));
  }
```

```
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
Test and Result Files Listings

The input file used in Section 3.3.3 and the resulting output are listed in this appendix. The reader may want to know that a value pair $< key, value >$ in the input/output files for HMR is represented as "key <tab> value," and each pair occupies a line. We also represent the value of $list < sourceURL >$ by "URL1|URL2|URL3|...". Each URL is concatenated with the "|" character if more than one URL is in the list.

D.1 Test Input

```
a	b
b	c
c	d
d	e
e	f
f	g
g	h
h	i
i	j
j	k
k	l
l	m
m	n
n	o
o	p
p	q
q	r
r	s
s	t
t	u
u	v
```

v w
w x
x y
y z
z a
aa b
bb c
cc d
dd e
ee f
ff g
gg h
hh i
ii j
jj k
kk l
ab b
bc c
cd d
de e
ef f
ab bb
bc cc
cd dd
de ee
ef ff
a zz
z yy
y xx
x ww
w vv
v uu
u tt

```
t   ss
s   rr
r   qq
a   zy
z   yx
y   xw
x   wv
w   vu
zy  c
yx  b
yy  b
xw  a
```

D.2 Test Result

```
a   c
aa  c
ab  c
b   d
bb  d
bc  d
c   e
cc  e
cd  e
d   f
dd  f
de  f
e   g
ee  g
ef  g
f   h
ff  h
g   i
```

```
gg  i
h   j
hh  j
i   k
ii  k
j   l
jj  l
k   m
kk  m
l   n
m   o
n   p
o   q
p   r
q   qq|s
r   rr|t
s   ss|u
t   tt|v
u   uu|w
v   vv|x|vu
w   ww|y|wv
x   xx|z|xw
xw  zz|b|zy
y   yy|a|yx
yx  c
yy  c
z   zz|b|zy
zy  d
```

APPENDIX E:
Sorting MR Source Listings

The sorting MR job source listings are contained in this appendix. The mapper returns the list of sources into separate value pairs, and the reducer sorts all the sources in alphabetical order.

E.1 Driver (SortDriver.java)

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class SortDriver extends Configured implements Tool{
  public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf, new SortDriver(), args));
  }

  @Override
  public int run(String[] arg0) throws Exception {
    // job1
    Configuration conf = this.getConf();

    Job job = Job.getInstance(conf, "SortDriver");
    // setup all classes

    job.setJarByClass(SortDriver.class);
```

```
    job.setMapperClass(SortMapper.class);
    job.setReducerClass(SortReducer.class);

    // input/output setup
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    // path setup
    String inputPath = arg0[0];
    String outputPath = arg0[1];

    FileInputFormat.addInputPaths(job, inputPath);
    FileOutputFormat.setOutputPath(job, new Path(outputPath));

    return job.waitForCompletion(true) ? 0 : 1;
  }
}
```

E.2 Mapper (SortMapper.java)

```
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class SortMapper extends Mapper<LongWritable, Text, Text,
   Text>{

  // Read from format: target <tab> source1|source2|source3...
  // to (target, source1), (target, source2), ...
  public void map(LongWritable key, Text value, Context context)
     throws IOException, InterruptedException{
    String[] token = value.toString().split("\t");
```

```
    String[] source = token[1].toString().split("\\|");
    for(String s: source)
      context.write(new Text(token[0]), new Text(s));
  }
}
```

E.3 Reducer (SortReducer.java)

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SortReducer extends Reducer<Text, Text, Text, Text>{

  // Read value pairs <target, source1>, <target, source2>, ...
  // output value pairs <target, list(source) where the list is
     in alphabetical order
  @Override
  public void reduce(Text key, Iterable<Text> values, Context
     context) throws IOException, InterruptedException {
    Iterator<Text> it = values.iterator();
    Set<String> sources = new HashSet<String>();

    // clean up repeated and null/empties
    while(it.hasNext())  sources.add(it.next().toString());
    sources.remove(Arrays.asList("", null));
```

```
        List<String> orderedSrc = new ArrayList<String>(sources);
        Collections.sort(orderedSrc); // sort the list here
        Iterator<String> it2 = orderedSrc.iterator();
        StringBuilder strBdr = new StringBuilder();

        // append the first value
        strBdr.append(it2.next());
        // append every other "| value" if there's any
        while(it2.hasNext()) strBdr.append("|").append(it2.next());

        // outputs the value pair
        context.write(key, new Text(strBdr.toString()));
    }
}
```

APPENDIX F:
JavaScript Object Notation (JSON) File Format

JSON [33] is a lightweight file format that is easy for humans to read and write and is being used in AWS. We used JSON files to save the environment variables in this study. As stated in Chapter 3 and Appendix A, there are seven files for Hadoop configuration, and we can rewrite part of each configuration file on EMR through a single JSON configuration file. A template of the JSON file for Hadoop configuration on EMR is provided here:

```
[
    {
        "Classification": "[XML File]",
        "Properties": {
            "[Variable Name1]": "[Value1]",
            "[Variable Name2]": "[Value2]"
        }
    },
    {
        "Classification": "[SH File]",
        "Properties": {},
        "Configurations": [
            {
                "Classification": "export",
                "Properties": {
                    "[Variable Name3]": "[Value3]"
                }
            }
        ]
    }
]
```

The JSON file always starts and ends with a pair of square brackets, and the configuration for different files is in a pair of curly brackets, separated by commas. There are two kinds of

files in those seven configuration files: XML and SH files. Each type of file must be used as shown in the template; only values in square brackets should be changed to the desired content. Misplaced commas are a common mistake. Make sure that commas are contained within the same brackets and that the last item in brackets is not followed by a comma.

APPENDIX G:
AWS EMR Error Message Track Down

In AWS, there are three different levels in which we can find an error message: cluster level, task level and container level. An example to illustrate the location of these error messages and how to track the container-level error message is used in this appendix.

G.1 Cluster-Level Error Message

The cluster-level error message is the most obvious one. It can be found in the AWS GUI as shown below:

1. First Method

   The first method to find a cluster-level error message is shown in Figure G.1 and Figure G.2.

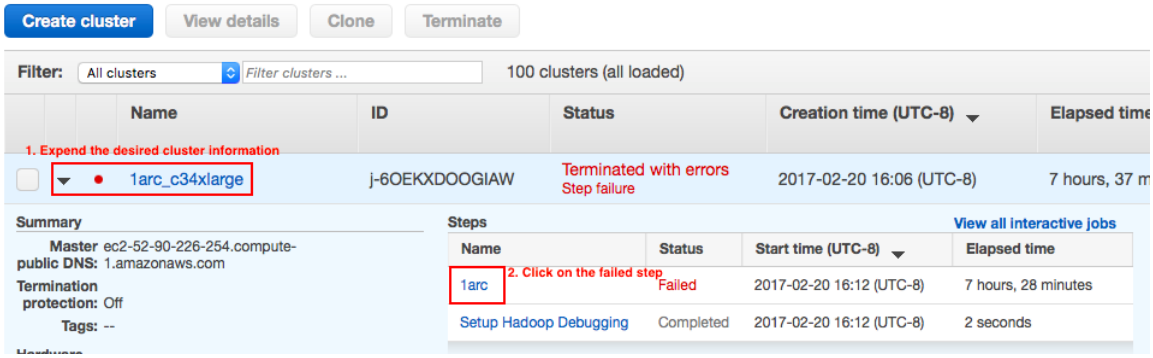


Figure G.1. First Method for Finding the Cluster-Level Error Message (1)

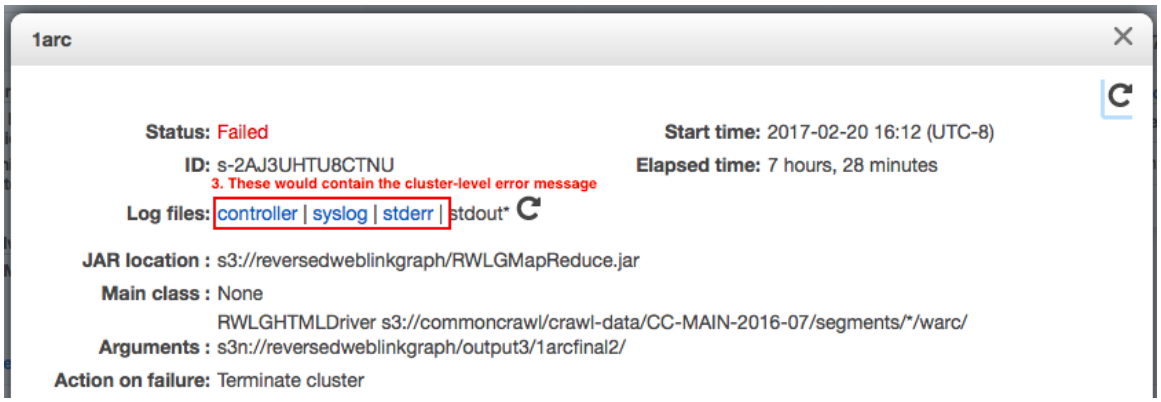


Figure G.2. First Method for Finding the Cluster-Level Error Message (2)

2. Second Method

   The second method to find a cluster-level error message is shown in Figure G.3 and Figure G.4.

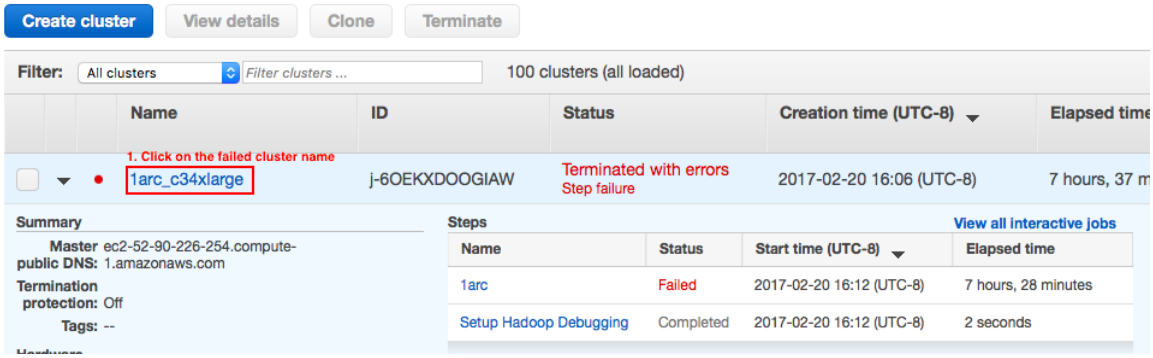


Figure G.3. Second Method for Finding the Cluster-Level Error Message (1)

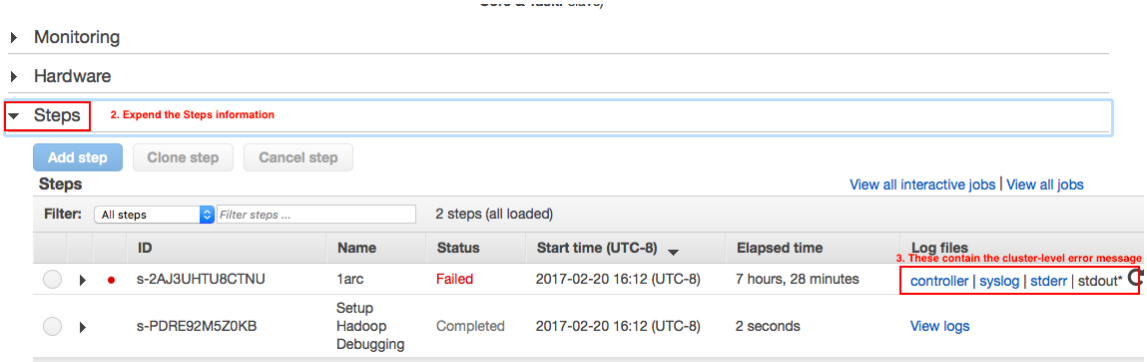


Figure G.4. Second Method for Finding the Cluster-Level Error Message (2)

3. Third Method

   The Third method to find a cluster-level error message is shown in Figure G.5, Figure G.6, and Figure G.7.

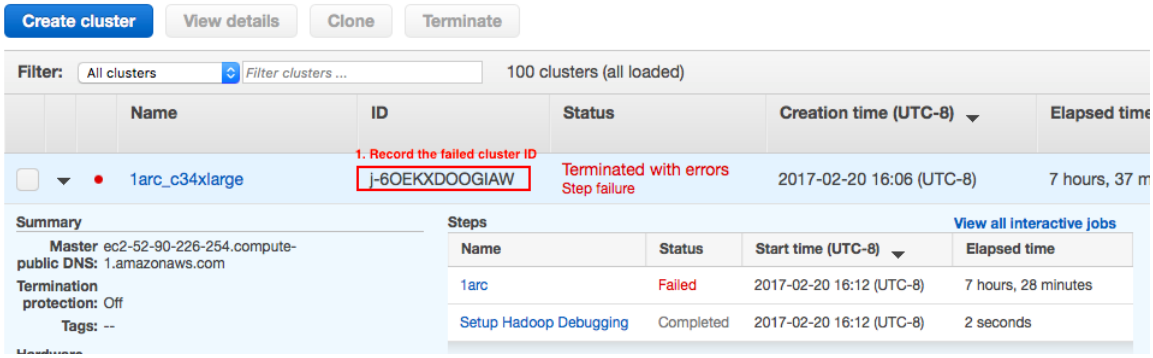


Figure G.5. Third Method for Finding the Cluster-Level Error Message (1)

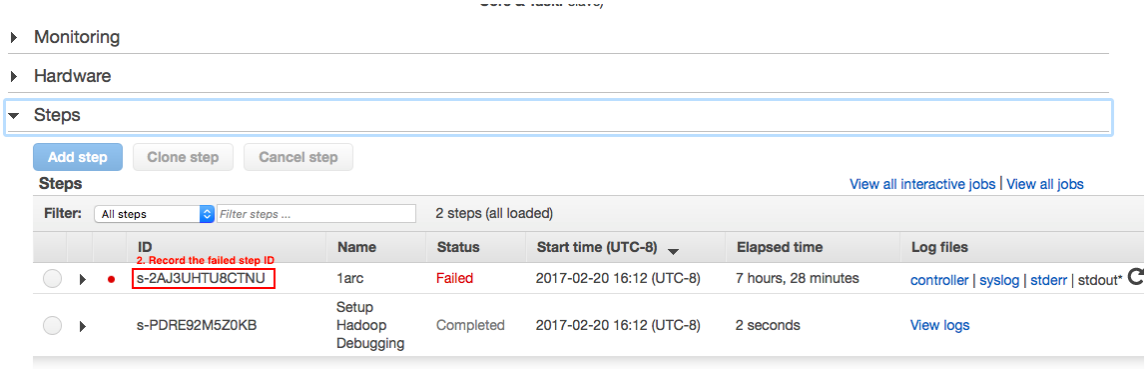


Figure G.6. Third Method for Finding the Cluster-Level Error Message (2)

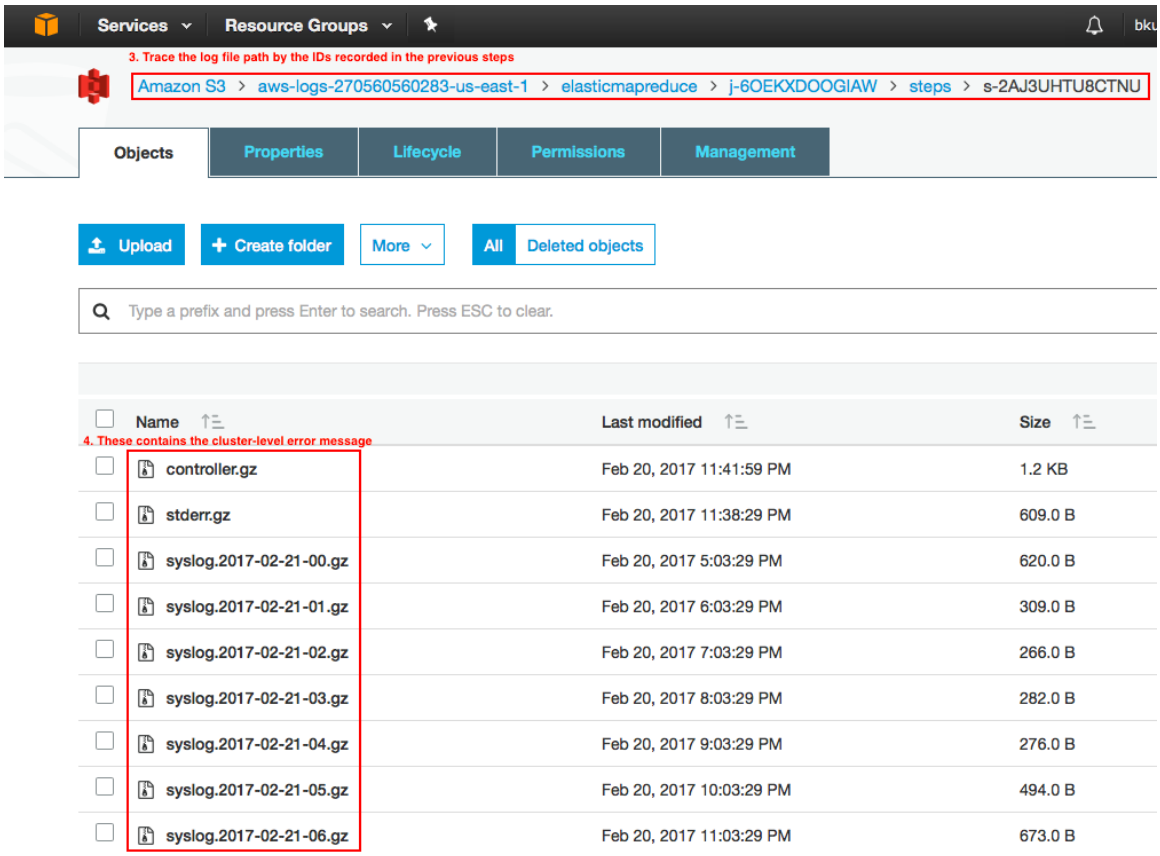


Figure G.7. Third Method for Finding the Cluster-Level Error Message (3)

G.2 Task-Level Error Message

1. First Method

   Continuing from the last step of the First Method to find the cluster-level error message, the first method to find a task-level error message is shown in Figure G.8, Figure G.9, and G.10.

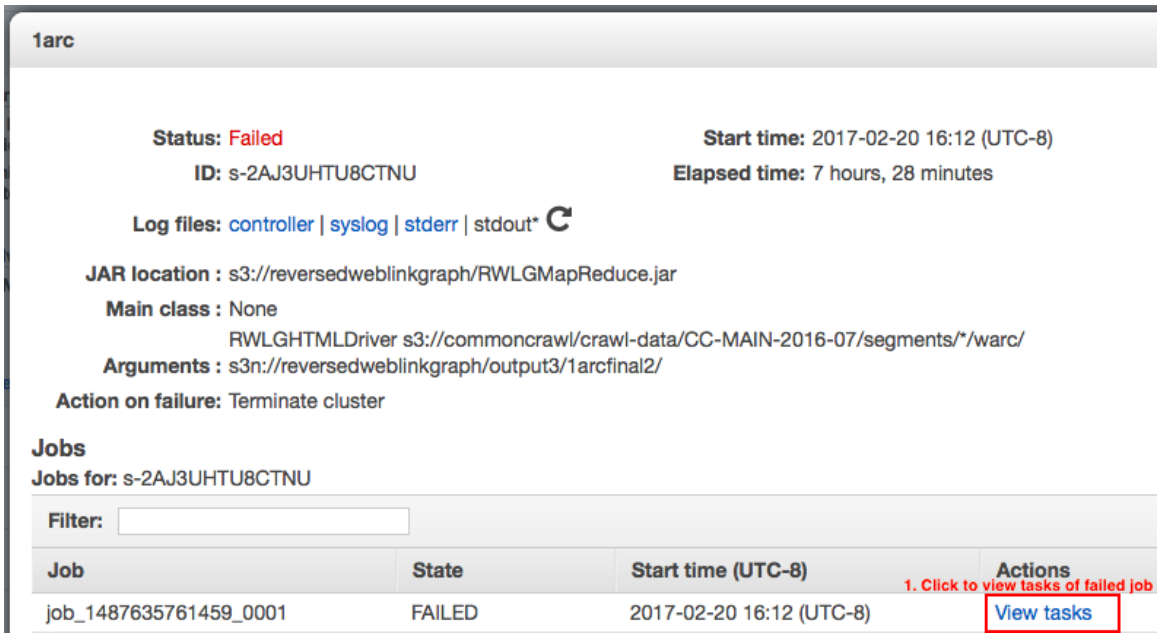


Figure G.8. First Method for Finding the Task-Level Error Message (1)

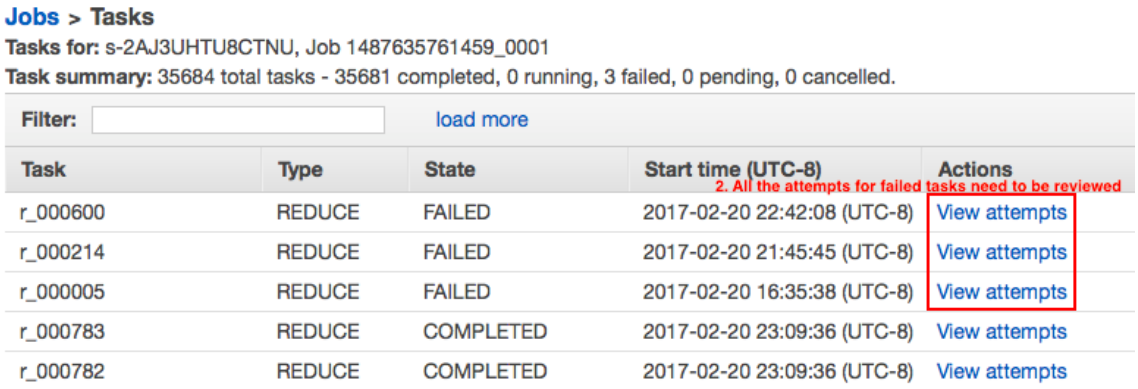


Figure G.9. First Method for Finding the Task-Level Error Message (2)

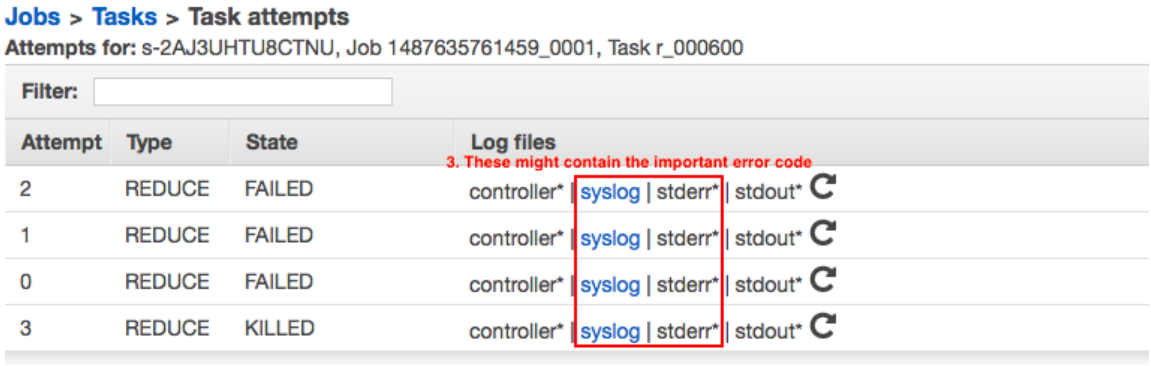


Figure G.10. First Method for Finding the Task-Level Error Message (3)

2. Second Method

   The second method also continues from the last step of the Second Method to find the cluster-level error message. It is shown in Figure G.11.

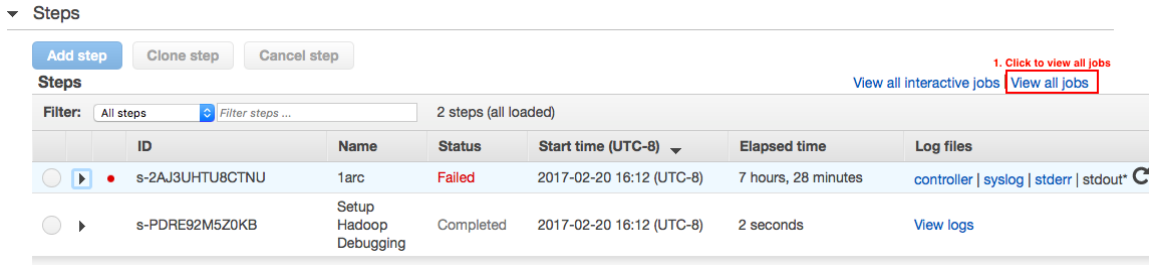


Figure G.11. Second Method for Finding the Task-Level Error Message

The rest of the Second Method is the same as the First Method.

G.3 Container-Level Log

It is rare to need to look in the container-level log for an error; however, when either a cluster-level or a task-level error message indicates the container number (as shown in Figure G.12), it is the time to look for the container log to trace the cause of the error.

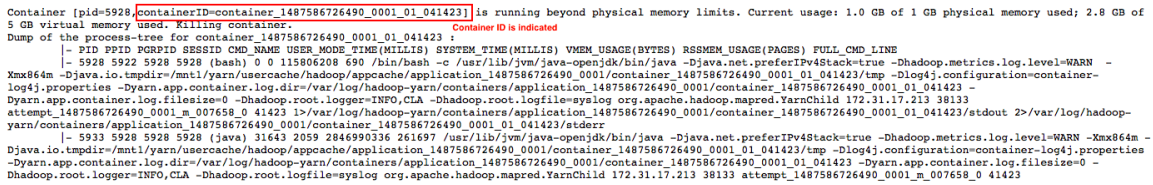


Figure G.12. Information about the Container-Level Log

The log files are all located in S3. We must go to the folder of the specified container, as shown in Figure G.13.

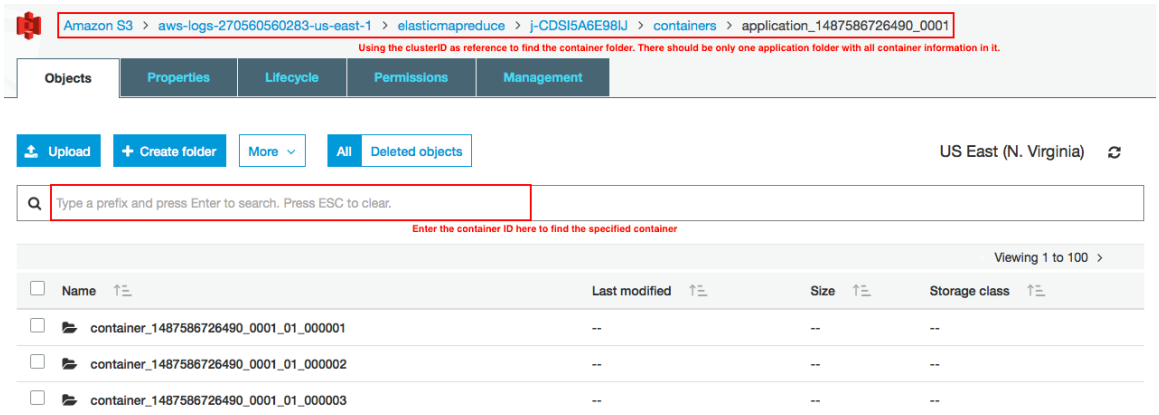


Figure G.13. How to Find the Specified Container Folder

After we get to the specified container, we are able to access the container-level log to sort out the problem in the failed execution, as shown in Figure G.14.

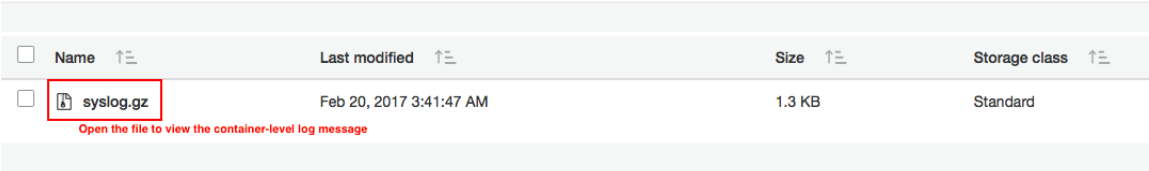


Figure G.14. Content of the Desired Container Folder

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. Kotenko. (2013, Sep. 18). Facebook reveals we upload a whopping 350 million photos to the network daily. *Digital Trends*. [Online]. Available: http://www.digitaltrends.com/social-media/according-to-facebook-there-are-350-million-photos-uploaded-on-the-social-network-daily-and-thats-just-crazy/

[2] A. Coudray, "Data mining of extremely large ad hoc data sets to produce inverted indices," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2016. Available: https://calhoun.nps.edu/handle/10945/49441

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*. New York, NY: ACM, 2013, pp. 5:1–5:16. Available: http://doi.acm.org/10.1145/2523616.2523633

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[6] Apache Hadoop YARN. (2016, Jan.). Apache Software Foundation. Los Angeles, CA. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html

[7] T. White, *Hadoop: The Definitive Guide, 4th Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2015.

[8] Determine HDP memory configuration settings. (2016, 11). Hortonworks. [Online]. Available: http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.5.3/bk_command-line-installation/content/determine-hdp-memory-config.html

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 2012, pp. 2–2.

[10] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," in *Proceedings of the VLDB Endowment*, no. 13. VLDB Endowment, 2015, vol. 8, pp. 2110–2121.

[11] S. Gopalani and R. Arora, "Comparing Apache Spark and Map Reduce with performance analysis using k-means," *International Journal of Computer Applications*, vol. 113, no. 1, 2015.

[12] J. Barr. (2013, May 14). Choosing the right EC2 instance type for your application. [Online]. Available: https://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application/

[13] S. Daneshyar and M. Razmjoo, "Large-scale data processing using mapreduce in cloud computing environment," *International Journal on Web Service Computing*, vol. 3, no. 4, p. 1, 2012.

[14] I. Finocchi, M. Finocchi, and E. G. Fusco, "Counting small cliques in mapreduce," *Computer Research Repository*, vol. abs/1403.0734, 2014. Available: http://arxiv.org/abs/1403.0734

[15] R. Mittal and R. Bagga. (2015, Oct.). Performance analysis of multi-node Hadoop clusters using Amazon EC2 instances. *International Journal of Science and Research*. [Online]. pp. 2319–7064. Available: https://www.ijsr.net/archive/v4i10/SUB159120.pdf

[16] N. Shrestha, R. K. Kharel, J. Britt, and R. Hasan, "High-performance classification of phishing urls using a multi-modal approach with mapreduce," in *Proc. IEEE World Congress on Services*, 2015, pp. 206–212.

[17] X. Lin, Z. Meng, C. Xu, and M. Wang, "A practical performance model for hadoop mapreduce," in *IEEE International Conference Cluster Computing Workshops*, 2012, pp. 231–239.

[18] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "A methodology for understanding mapreduce performance under diverse workloads," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-135, Nov 2010. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-135.html

[19] Amazon EMR pricing. (2017, Feb. 7). Amazon Web Services. [Online]. Available: https://aws.amazon.com/emr/pricing/

[20] Amazon EC2 pricing. (2017, Feb. 7). Amazon Web Services. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[21] Amazon S3 pricing. (2017, Feb. 7). Amazon Web Services. [Online]. Available: https://aws.amazon.com/s3/pricing/

[22] *Information and documentation — WARC file format*, ISO Standard 28500:2009(en), 2009.

[23] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform resource locators (URL)," Internet Requests for Comments, RFC Editor, RFC 1738, December 1994. Available: http://www.rfc-editor.org/rfc/rfc1738.txt

[24] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," Internet Requests for Comments, RFC Editor, STD 66, January 2005. Available: http://www.rfc-editor.org/rfc/rfc3986.txt

[25] M. Duerst and M. Suignard, "Internationalized resource identifiers (IRIs)," Internet Requests for Comments, RFC Editor, RFC 3987, January 2005. Available: http://www.rfc-editor.org/rfc/rfc3987.txt

[26] S. Rhodes, "Breaking down link rot: the Chesapeake project legal information archive's examination of URL stability," *Law Library Journal*, vol. 102, p. 581, 2010.

[27] G. Popoff. (2014, Apr. 21). Big data and hadoop. [Online]. Available: http://glebche.appspot.com/static/hadoop-ecosystem/hadoop-hive-tutorial.html#hadoop-installation-mac-osx

[28] Marek. (2014, Sep. 23). Installing Hadoop on Mac part 1. [Online]. Available: https://amodernstory.com/2014/09/23/installing-hadoop-on-mac-osx-yosemite/

[29] Hive Java heap error running query (exit code 143). (n.d.). Cloudera Community. [Online]. Available: http://community.cloudera.com/t5/Batch-SQL-Apache-Hive/Hive-Java-heap-error-running-query-exit-code-143/td-p/8668. Accessed: Nov. 20, 2016.

[30] Task configuration. (2017, Feb. 7). Amazon Web Services. [Online]. Available: http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-task-config.html

[31] Hadoop daemon settings. (2017, Feb. 7). Amazon Web Services. [Online]. Available: http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-daemons.html

[32] V. Goel. (2016, Oct. 23). Defining web pages, web sites and web captures. [Online]. Available: https://blog.archive.org/2016/10/23/defining-web-pages-web-sites-and-web-captures/

[33] ECMA, *ECMA-404: The JSON Data Interchange Format*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), October 2013, 2013. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California